FORSCHUNGSZENTRUM JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik
D-52425 Jülich, Tel. (02461) 61-6402

Interner Bericht

Beiträge zum Wissenschaftlichen Rechnen
Ergebnisse des
Gaststudentenprogramms 2004
des John von Neumann-Instituts
für Computing

Rüdiger Esser (Hrsg.)

FZJ-ZAM-IB-2004-11

November 2004

(letzte Änderung: 9. 11. 2004)

Vorwort

Die Ausbildung im Wissenschaftlichen Rechnen ist neben der Bereitstellung von Supercomputer-Leistung und der Durchführung eigener Forschung eine der Hauptaufgaben des John von Neumann-Instituts für Computing (NIC) und hiermit des ZAM als wesentlicher Säule des NIC. Um den akademischen Nachwuchs mit verschiedenen Aspekten des Wissenschaftlichen Rechnens vertraut zu machen, führte das ZAM in diesem Jahr zum fünften Mal während der Sommersemesterferien ein Gaststudentenprogramm durch. Entsprechend dem fächerübergreifenden Charakter des Wissenschaftlichen Rechnens waren Studenten der Natur- und Ingenieurwissenschaften, der Mathematik und Informatik angesprochen. Die Bewerber mussten das Vordiplom abgelegt haben und von einem Professor empfohlen sein.

Die neun vom NIC ausgewählten Teilnehmer kamen für zehn Wochen, vom 2. August bis 8. Oktober 2004, ins Forschungszentrum. Sie beteiligten sich hier an den Forschungs- und Entwicklungsarbeiten des ZAM und wurden jeweils einem Wissenschaftler zugeordnet, der mit ihnen zusammen eine Aufgabe festlegte und sie bei der Durchführung anleitete.

Die Gaststudenten und ihre Betreuer waren:

Nikos Elpidoforou	Paul Gibbon
Ivo Kabadshow	Holger Dachsel
Slawomir Pitula	Thomas Müller
Nikolas Pomplun	Guido Arnold, Marcus Richter
Armin Rund	Bernhard Steffen
Sebastian Schiffner	Herwig Zilken
Jakob Schluttig	Godehard Sutmann
Jiulong Shan	Volker Sander
Jonas Wiebke	Thomas Müller

Zu Beginn ihres Aufenthalts erhielten die Gaststudenten eine viertägige Einführung in die Programmierung und Nutzung der Parallelrechner im ZAM. Um den Erfahrungsaustausch untereinander zu fördern, präsentierten die Gaststudenten am Ende ihres Aufenthalts ihre Aufgabenstellung und die erreichten Ergebnisse. Sie verfassten zudem Beiträge mit den Ergebnissen für diesen Internen Bericht des ZAM.

Dieser Band enthält zusätzlich den Arbeitsbericht von Benjamin Sohn, der - betreut von Inge Gutheil - während der Sommer-Semesterferien ein längeres Praktikum im ZAM durchgeführt hat.

Wir danken den Teilnehmern für ihre engagierte Mitarbeit - schließlich haben sie geholfen, einige aktuelle Forschungsarbeiten weiterzubringen - und den Betreuern, die tatkräftige Unterstützung dabei geleistet haben.

Ebenso danken wir allen, die im ZAM und in der Verwaltung des Forschungszentrums bei Organisation und Durchführung des diesjährigen Gaststudentenprogramms mitgewirkt haben. Besonders hervorzuheben ist die finanzielle Unterstützung durch den Verein der Freunde und Förderer des FZJ und die Firma IBM. Das erfolgreiche Programm soll auch in den kommenden Jahren weitergeführt werden, schließlich ist die Förderung des wissenschaftlichen Nachwuchses dem Forschungszentrum ein besonderes Anliegen.

Weitere Informationen über das Gaststudentenprogramm, auch die Ankündigung für das kommende Jahr, findet man unter http://www.fz-juelich.de/zam/gaststudenten.

Jülich, November 2004 Rüdiger Esser

Inhalt

Fast Near-Neighbour Search in Molecular Dynamics Simulation Using a Hashed Oct Tree

Nikos Elpidoforou

University of Athens, Greece,
Department of Chemistry,
Laboratory of Physical Chemistry

E-mail: fi nwue@yahoo.com

Abstract:

The traditional near-neighbour searches in molecular dynamics codes require an O(N) computational effort per particle. Tree Algorithms can reduce this to O(logN) exploiting the tree structure to obtain an initial list of neighbour 'boxes'. The aim of this article is to explore a fast near-neighbour search algorithm based on the Hashed Oct Tree data structure. The particle coordinates are mapped onto a sorted list of binary keys, which can be used to rapidly determine the location of the particles within the tree. The introduced algorithm is exploiting the properties of the particle keys to create a near-neighbour list.

Introduction

The basic role of a computer simulation is to provide us with approximate solutions for a series of problems concerning Statistical Mechanics or any other domain of science where we need to study a set of N interacting objects. This challenge is well known as the N-body problem. Simulation can be either a test of theory or a test of models. It is because of that particular connection that it is often called 'Computer Experiment'. In the field of Statistical Mechanics the simulation gives us a straightforward connection between the microscopic and the macroscopic properties of a system of N molecules. In other words through some simulation techniques we can obtain properties of experimental interest (e.g. transport coefficients) from the molecular details of a particular system. One of those techniques is Molecular Dynamics [1]. We will give a short overview of the method in the following chapter.

Molecular Dynamics

Molecular Dynamics (MD) is the term we use to describe the solution of the classical equations of motion for a set of N molecules. The aim is to generate the molecular trajectories as accurately as possible. Considering a set of N interacting molecules with Cartesian coordinates r_i and the usual definitions of kinetic and potential energy the equations of motions can take a quite simple and familiar form:

$$m\ddot{r}_i = f_i \tag{1}$$

This shows that MD involves the quite difficult task of solving a system of 3N second order differential equations. A standard method that helps our cause is the finite difference approach which has two basic algorithmic forms:

The Gear Predictor-Corrector and the Verlet algorithm

The general idea of these approaches is to try to get the positions, velocities etc at a certain time t with a desired degree of accuracy using the positions, velocities etc of a previous time t-dt before. In this way the equations of motion can be solved on a step-by-step basis. The basic formulas of the two algorithms mentioned above are given in brief in the next sections.

Finite difference methods

The Verlet algorithm

The Verlet algorithm is based on the positions r(t) and the accelerations a(t) of a certain time step and the positions r(t-dt) of the previous step. So, in order to get the position of the next time step r(t+dt) we use the following equation:

$$r(t+\delta t) = 2r(t) - r(t-\delta t) + \delta t^2 \frac{1}{m} f(t) \tag{2}$$

Having the new positions, we can estimate the new accelerations through the forces that are acting over all molecules and thus move the whole system forward for another time step. We observe that the Verlet algorithm is time-reversible and that the velocities do not appear in the basic Eq. 2. The velocities are given by the next equation:

$$v(t) = \frac{r(t+\delta t) - r(t-\delta t)}{2\delta t} \tag{3}$$

The reason that they do not show up is that they are being eliminated by addition of the next equations.

$$r(t+\delta t) = r(t) + \delta t v(t) + \frac{1}{2}\delta t^2 \frac{1}{m} f(t) + ... \tag{4}$$

$$r(t-\delta t) = r(t) - \delta t v(t) + \frac{1}{2}\delta t^2 \frac{1}{m} f(t) - ... \tag{5}$$

Eqs.(4-5) are the Taylor expansions about r(t). There are several other forms of the basic Verlet algorithm that have been proposed to deal with problems that occur due to numerical imprecision. The most famous is the half step leap-frog algorithm that has the following form:

$$r(t+\delta t) = r(t) + \delta t v(t + \frac{1}{2}\delta t) \tag{6}$$

$$v(t+\frac{1}{2}\delta t) = v(t-\frac{1}{2}\delta t) + \delta t \frac{1}{m} f(t) \tag{7}$$

It is obvious that if we eliminate the velocities we will get the basic scheme of Verlet algorithm which is showing us that the two forms are mathematically equivalent.

The Gear Predictor-Corrector

The Gear predictor-corrector algorithm tries to generate the trajectories of the molecules obtaining first an estimate of the positions, velocities etc at time t+dt. The equations for such a prediction are given below:

$$r^p(t+\delta t) = r(t) + \delta t v(t) + \frac{1}{2}\delta t^2 a(t) + \frac{1}{6}\delta t^3 b(t) + ... \tag{8}$$

$$v^p(t+\delta t) = v(t) + \delta t a(t) + \frac{1}{2}\delta t^2 b(t) + ... \tag{9}$$

$$a^p(t+\delta t) = a(t) + \delta t b(t) + ... \tag{10}$$

$$b^p(t+\delta t) = b(t) + ... \tag{11}$$

Afterwards we apply a correction step. That is, we calculate the 'correct' new accelerations from the predicted values of the new positions. This is done by evaluating the forces over all molecules. The 'correct' new accelerations are compared with the 'predicted' new accelerations and we obtain an estimate of the error in the prediction step. The comparison is performed with the following equation:

$$\Delta a(t+\delta t) = a^c(t+\delta t) - a^p(t+\delta t) \tag{12}$$

The error we calculate in Eq.(12) is used to correct the predicted values as we can see through the next equations:

$$r^c(t+\delta t) = r^p(t+\delta t) + c_0 \Delta a(t+\delta t) \tag{13}$$

$$v^c(t+\delta t) = v^p(t+\delta t) + c_1 \Delta a(t+\delta t) \tag{14}$$

$$a^c(t+\delta t) = a^p(t+\delta t) + c_2 \Delta a(t+\delta t) \tag{15}$$

$$b^c(t+\delta t) = b^p(t+\delta t) + c_3 \Delta a(t+\delta t) \tag{16}$$

The corrected values obtained by Eqs.(13-16) are better approximations to the real values of the positions, velocities etc. If we want we can repeat the correction step in order to achieve a further refinement to the obtained values. The coefficients in Eqs.(13-16) depend on the order of the differential equation being solved and many different sets of values have been proposed [1].

Neighbour lists

The algorithms presented in the two previous sections show that the most crucial part in a MD simulation is where we evaluate the forces and hence the new accelerations obtained from the new positions of the molecules. That is true as long as we consider that the greatest amount of computational time is spent examining the set of the pairs of molecules and identifying those pairs separated by less than a given radius and computing the forces for this subset. For each molecule the set of neighbours within this specific radius changes with time and the task of identifying and rejecting molecules is very time consuming. Our goal is to reduce the time needed to build a certain near-neighbour list using a Hashed Oct-Tree data structure. In the next sections we will present the basic neighbour listing techniques that are used and the basics of the Hashed Oct Tree algorithm.

We have already mentioned the presence of a certain radius around a molecule. That radius is actually the distance up to where we consider that this molecule can 'sense' the presence of a different one due to their pairwise interaction. If a molecule lies outside of this radius we do not take into account any interaction with the certain one. This radius is called the potential cut-off. Physically this is justified because the interactions at great distances between neutral molecules are negligible. A quite simple approach in MD simulation is for a certain molecule i to loop over all molecules j searching for the ones which lie inside the cut-off. This is very time consuming proportional to N^2 and it slows down our simulation. To avoid this there are two methods of building neighbour lists for each molecule. The Verlet algorithm and the cell structures and linked lists method [1].

The Verlet Algorithm

The idea behind this algorithm is to create a thick 'skin' around the cut-off sphere of a molecule. The molecule is now sitting in the centre of two spheres at the same time. The one sphere has a radii equal to the cut-off and the second sphere has a radii r(l) which is greater than the cut-off. Then a large array NEIGHBOUR is constructed which contains all of the neighbours of each molecule inside the r(l). In order to find the neighbours of a given molecule we need an index and that is satisfied by an another array INDEX which points to the position of the array NEIGHBOUR where the first neighbour of the given molecule lies. The code locates the neighbours of a molecule (i) by checking the array NEIGHBOUR from the position INDEX(i) to INDEX(i+1)-1. This neighbour list is then updated at intervals of 10-20 timesteps.

Cell structures and linked lists

The previous neighbour listing fails when we are dealing with sets of more than 1000 molecules. The size of the array NEIGHBOR becomes too large and creates storage problems. So an alternative method has been proposed: the cell index method. We divide the simulation box into a lattice of cells. The side of the cell is greater than the cut-off radius. We create a separate list of molecules in each of those cells and we can speed up the neighbouring search checking only the cells we are interested in. The whole procedure is carried out by the method of linked lists. We first sort the molecules into the cells while we create two arrays. The first array HEAD has one element for each cell that contains the identification number for one of the molecules in that cell. This number is used to address the element of the second array LIST which has the number of the next molecule in that cell. After that the element for that molecule is the index of the next molecule in the cell and so on. Finally after several indexing numbers of molecules we will reach an element of LIST which is zero. That means that we have checked all the molecules of this cell and we can move forward to the next cell through the array HEAD.

The Hashed Oct Tree Algorithm (HOT)

Introduction

One of the motivations for using a tree data structure for the neighbour searching problem is that we have the opportunity to reject large sets of molecules or particles quickly and easily. That is due to the oct-tree data structure that 'distributes' the particles into cubical regions called from now on 'cells'. There is a great cube, representing the whole simulation box, called the 'root'. We can divide each of the dimensions of the root into half and produce eight subcells inside the root. This 'cutting' procedure is continued for every subcell, until each cell is either empty or contains only one particle. This operation can be optimised by constructing binary keys to map the 3-dimensional spatial coordinates of the particles onto a one dimensional curve. The keys do not replace the particles' coordinates, but give us a rapid means of sorting them and building the oct-tree structure around them. The keys are constructed from the following operation, if we are referring to an oct-tree (3d):

$$\text{key} = \text{placebit} + \sum_{j=0}^{\text{nbits}-1} 8^j \left(4 \times \text{BIT}(i_z, j) + 2 \times \text{BIT}(i_y, j) + \text{BIT}(i_x, j)\right) \tag{17}$$

The function BIT ( ) selects the j-bit of the integer coordinate components which are given from:

$$i_x = \frac{x}{s}, \qquad i_y = \frac{y}{s}, \qquad i_z = \frac{z}{s} \tag{18}$$

where:

$$s = \frac{L}{2^{\text{nlev}}} \tag{19}$$

and L is the length of the simulation box. Actually nbits is the length of the dimensions of the 'root'. nlev is the maximum refinement level. That means that s is referring to the length of the dimensions of the subcells in the maximum refinement level. The placebit is defined by:

$$\text{placebit} = 2^{D \times \text{nlev}} \tag{20}$$

where D=3.

We add this placebit to avoid any ambiguity among keys at different levels. This placebit is also called the place-holder bit and represents the 'root'. The construction of the keys, if we are referring to a quad-tree (D=2) is the same:

$$\text{key} = \text{placebit} + \sum_{j=0}^{\text{nbits}-1} 4^j \left(2 \times \text{BIT}(i_y, j) + \text{BIT}(i_x, j)\right) \tag{21}$$

The parameters are defined the same way like in Eqs.(18-20). Figure 1 shows how a tree structure is built in 2 dimensions.

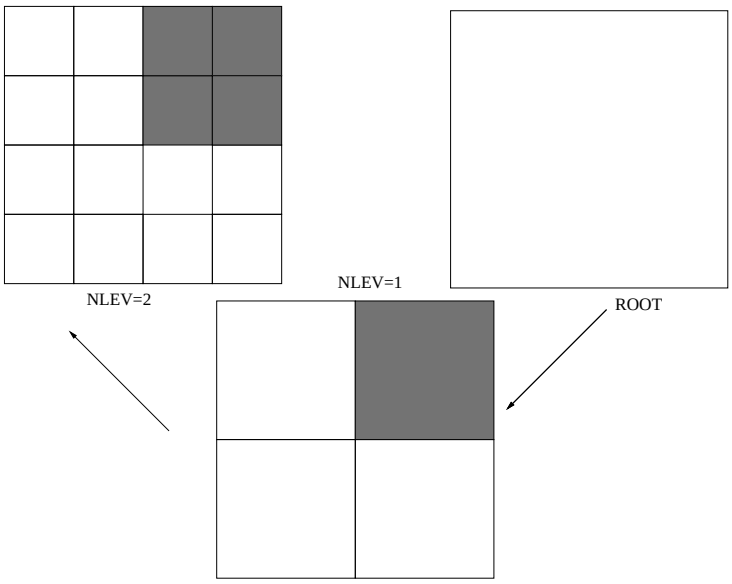


Figure 1: Tree structure in 2 dimensions.

All particles are first attached to the 'root'. The 'root' is divided into 4 sub-squares and the particles re-attached accordingly. A sub-square containing only one particle is defined as a 'leaf', while sub-squares with more than one are defined as 'twigs' and subsquares with no particles are discarded. This procedure is continued until each particle sits in its own square. Each key identifies the location in the tree of a particular collection of data. To retrieve the data corresponding to any key, we must translate the key to a pointer that shows the memory location containing the data. So, obviously an indexing problem arises because of the very large number of possible keys. To overcome this, we map the values from the very large set of possible keys into a smaller set which is used as an index into a hash table. This mapping is called hashing. For more details about the Hashed Oct Tree algorithm the reader can refer to [2].

The HOT Neighbour Search

We have already mentioned that we want to build near-neighbour lists exploiting the tree structure and basically the properties of the particle keys. The initial goal is to invent a fast bit operation in order to find the neighbouring keys of a specific particle key. This key is actually representing a sub-cell (a box) in the tree structure and may contain more than one particle. So we will try to find a way of calculating the neighbouring sub-cells from the given key (sub-cell). This is clear in Fig. 2 where we give a 2D example. We consider a maximum refinement level of 2 and for a given particle we find its key (sub-square) and try to calculate from that the neighbouring keys (sub-squares).

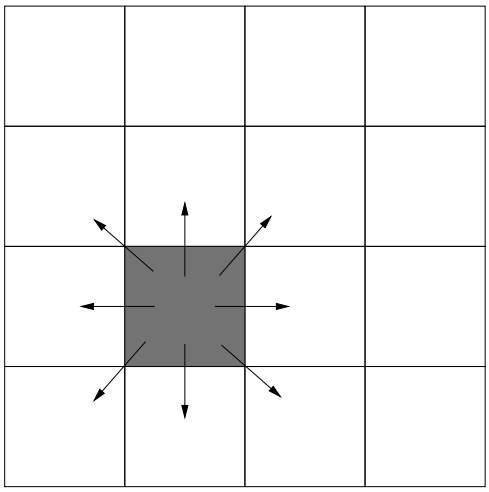


Figure 2: 2D example.

Moving inside the same parent box

The construction of the tree structure deals with a very simple operation; getting the parent and the child keys of a particle key in a specific level of refinement. The parent and the child keys can be found by simple bit shifting operations [3] and represent the directly lower and higher levels of refinement respectively. So, for a given level of refinement we have to consider first how to move inside the same parent box. We will consider a 2 dimensional example (Fig.3) to illustrate our discussion.

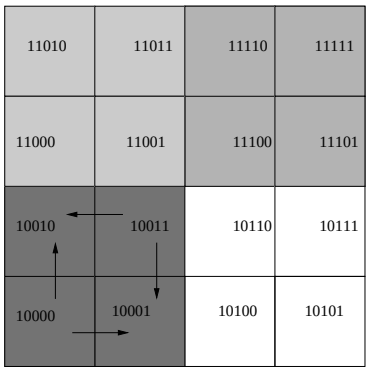


Figure 3: Moving inside the same parent.

When we want to get the square that lies on the right of the given square we will use the phrase; "to

move right". Equal expressions will be used to get the squares that lie on the left and so on. Moving right inside the same parent box means that we have to add one (+1) to the key and moving left means we need to substract one (-1). Moving up we add two (+2) and moving down we substract two (-2). Diagonally movement can be achieved with linear combination of those two simple operations. So, a first form of neighbour searching can be the following:

$$K(ij) = particle\_key + c(i) + c(j) \tag{22}$$

where c(i) determines the movement on the x direction and c(j) determines the movement on the y direction.

Those two components are given by the next equations.

$$c(i) = |i|\ sign(i)\ ,\ i=-1,0,1 \tag{23}$$

$$c(j) = 2\ |j|\ sign(j)\ ,\ j=-1,0,1 \tag{24}$$

Crossing the boundaries of the parent boxes

A problem arises when a neighbouring box lies in the neighbouring parent box (Fig.4). So, in order to get it we have to cross the boundaries of the parent box to which our particle key belongs. As one can see, crossing a right boundary we need to add three (+3) and crossing the left one we have to substract three (-3). A top boundary forces us to add six (+6) and the opposite make us substract six (-6). That means we must think of a way to extend the Eqs. (23-24) in the case of crossing the boundaries of the parent boxes. We formalise this procedure with the following logic:

Moving rightwards we have to add one or three.

```
c = 1 AND same-parent OR 3 AND not-same-parent
c = 1 x sp + 3 x nsp
c = 1 x (1-nsp) + 3 x nsp
```

$$c = 1 + 2 \times nsp \tag{25}$$

We consider sp and nsp being logical variables which take the values 1 or 0 if they are true or false respectively. Proceeding the same way for moving left we end up in the following.

$$c = -1 - 2 \times nsp \tag{26}$$

How can we replace the logical variable nsp which is only known *a posteriori*? We can do this by setting

$$nsp = BIT\ (particle\_key,\ 0) \tag{27}$$

when we move right and

$$nsp = 1 - BIT\ (particle\_key,\ 0) \tag{28}$$

when we move left. This bit operation is just picking the 0 bit of the particle key. If this is 0 we know it is at the left edge of the parent box; if it is 1, we know it is at he right edge of the parent box. So, if we substitute Eqs.(27),(28) into (25),(26) we end up with:

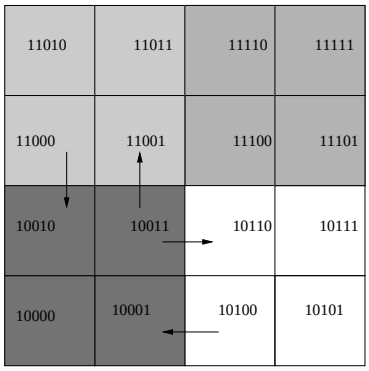


Figure 4: Crossing the boundaries of the parent boxes.

Right

$$c = 1 + 2*BIT(particle\_key, 0) \tag{29}$$

Left

$$c = -3 + 2*BIT(particle\_key, 0) \tag{30}$$

We must combine Eqs.(29)(30) to get a general form for the movement in the x direction. This combination gives us the next equation:

$$c(i) = |i|\ sign(i)\{1 + 2[\ w(i) + sign(i)BIT(particle\_key,0)]\} \tag{31}$$

where

$$i = -1,0,1 \text{ and } w(i)=(1-i)/2$$

When we use i=1 we end up in Eq.(29) that is valid for moves to the right. When we use i=-1 we end up in Eq. (30) that is valid for moves to the left. The value i=0 does not move the particle in the x direction. Extending this argument to the y direction we end up with a quite similar equation to Eq.(31).

$$c(j) = 2\ |j|\ sign(j)\{1 + 2[\ w(j) + sign(j)BIT(particle\_key,1)]\} \tag{32}$$

where

$$j = -1,0,1 \text{ and } w(j)=(1-j)/2$$

The only difference between Eqs.(31) and (32) is that we check the 2nd bit and that we multiply by two. But what happens when we have to cross boundaries of grand-parent boxes and so on? Clearly we must generalise the introduced Eqs. (31)(32) for any given level of refinement.

Crossing the boundaries of the grand-parent boxes-General formulation

We can see in Figure 5 the problem arising when crossing the boundaries of a grand-parent box. We see that to cross boundaries on the right one has to add eleven (+11) and on the left one has substracts eleven (-11). These increments are doubled for the y direction. Thus moving to the right is now expressed by:

$$c = 1 + 2 \times nsp + 8 \times nsgp \tag{33}$$

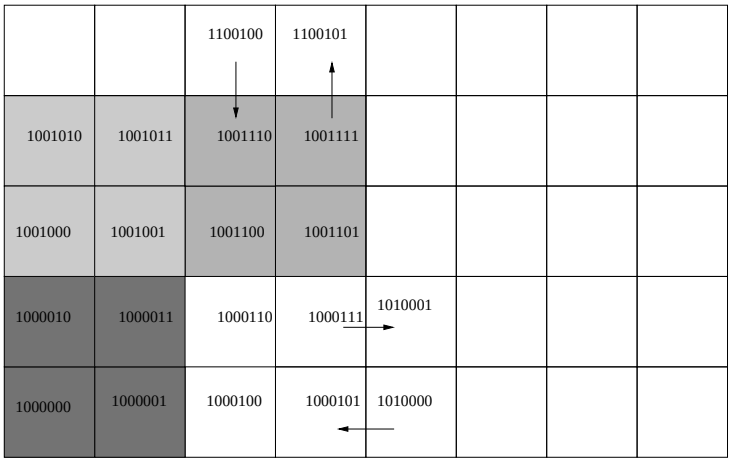


Figure 5: Crossing the boundaries of the grand parent boxes.

and moving to the left is:

$$\texttt{c = -1 - 2 x nsp - 8 x nsgp} \tag{34}$$

where nsgp signifies that we have crossed a grand-parent bounding.

We recall that:

$$\text{nsp} = \begin{cases} \text{BIT}(\text{K}, 0) & \text{(right)} \\ 1 - \text{BIT}(\text{K}, 0) & \text{(left)} \end{cases} \tag{35}$$

and one can show by inspection that:

$$\text{nsgp} = \begin{cases} \text{BIT}(\text{K}, 2) \times \text{BIT}(\text{K}, 0) & \text{(right)} \\ (1 - \text{BIT}(\text{K}, 2)) \times (1 - \text{BIT}(\text{K}, 0)) & \text{(left)} \end{cases} \tag{36}$$

We must ensure through Eqs.(36) that the box we are examining is placed on the boundaries of the grand parent box. The algebra involved with replacing in Eqs.(33),(34) the Eqs.(35),(36) is simple and we end up to the following for the x dimension.

$$c_i = |i| s_i \{1 + 2[w_i + s_i \times \text{BIT}(\text{K}, 0)] \times [1 + 4(w_i + s_i \times \text{BIT}(\text{K}, 2))]\} \tag{37}$$

The same logic leads us to the equation for the y dimension:

$$c_j = 2|j| s_j \{1 + 2[w_j + s_j \times \text{BIT}(\text{K}, 1)] \times [1 + 4(w_j + s_j \times \text{BIT}(\text{K}, 3))]\} \tag{38}$$

The similarities and differences are obvious and simple. We can continue the procedure when we cross great grandparent boxes and build new equations for any given higher level. Those equations will contain the already mentioned characteristics for the lower levels but can also give us the numbers needed crossing the highest boundaries at the refinement level we are. The general formula for any given level is finally the following:

$$c_i = \ |i| s_i \left[1 + \sum_{\lambda=1}^{\text{nlev-1}} 2^{2\lambda-1} \prod_{\phi=1}^{\lambda} [w_i + s_i \times \text{BIT}(\text{K}_{00}\,, 2\phi - 2)] \right] \tag{39}$$

$$c_j = 2|j| s_j \left[1 + \sum_{\lambda=1}^{\text{nlev-1}} 2^{2\lambda-1} \prod_{\phi=1}^{\lambda} [w_j + s_j \times \text{BIT}(\text{K}_{00}\,, 2\phi - 1)] \right] \tag{40}$$

Implementation

Coding the algorithm

We have used Fortran-90 programming language to code the algorithm of the neighbour searching in 2 dimensions. The code has two inputs: the number of particles and the search radius. The length of the 2 dimensional simulation box is fixed as is fixed the maximum number of particles. The maximum level of refinement we reach is defined by the search radius inside the code. We give random coordinates to the particles and we construct the particles keys for the maximum refinement level as we have already explained. That means that every particle has a key which is pointing to a box inside the root. We perform a simple sorting of the particles inside every box just before the neighbour searching starts. After that we are ready to start the searching. We pick one particle for which we know its particle key. Therefore we know inside which box it is sitting. Next, we apply the bit operations we have just introduced and find all the neighbouring boxes. We can now check which of the particles that are contained in the neighbouring boxes are inside the search radius of the given particle. The core part of the algorithm is given in the Appendix.

Time scaling

Is it working properly? This is the first question that arises. We must check if the algorithm is finding the correct neighbours or producing mistakes. The checks on several particles for different radi proves that it does indeed find the neighbours correctly. We show five examples of this below. Figure 6 has a 2 dimensional simulation box which contains 1000 particles represented as dots. The cross represents a randomly chosen particle. In figures 7 and 8 we can see the neighbours of this particle in a radius 5, 10, 15 and 20 per cent of the simulation box length respectively. These neighbours have been identified correctly by our algorithm.

The next question that arises concerns the time scaling of the algorithm. We have measured the time needed for our method and the time needed for the naive method looping over all particles j for a specific particle i. We have made time measurements for the already given radius and for a number of particles that vary from 1000 up to 8000. We can see through the figures 9 and 10 that the time needed for the naive method does not depend on the search radius, as expected and shows a time scaling proportional to N^2. Our new algorithm on the other hand has a strong dependency on the search radius. For short radius compared to the simulation box length it is much faster than the classical search and it is proportional to N. As we grow the search radius our method is slowing down and loses its linear dependency to the number of particles due to the increase in box size.

Conclusions

We have investigated a new algorithm for the fast near-neighbour search in MD simulation using a tree data structure. The algorithm has satisfying time scaling for short search radius. Further optimisation is necessary in order to speed up the neighbour searching in longer distances. This could be possible through further refinement of the boxes inside the search radius and thus automatically capturing a large proportion of particles that lie by definition inside the given radius. In that case we just need to search through a number of boxes that contain a small amount of particles and lie on the edges of the circle defined by our radius. We can also find a way to replace some time consuming intrinsic functions we have applied in the detailed search algorithm inside every box. During this project we have found the generalisations to the Eqs.(39),(40) that should be applied in a 3 dimensional simulation box and make the algorithm valid for 3 dimensional neighbour searching; this will be tested in future work. Finally, we note that a detailed time comparison between our algorithm and the cell index method has still to be performed.

Acknowledgements

This work was carried out during the Guststudents' Program for Scientific computing 2004, organised by ZAM and NIC. I would like to thank the organiser Dr. R. Esser for enabling me to participate in the program and all staff of ZAM for excellent working conditions. I would also like to thank my supervisor Dr. Paul Gibbon for his contribution to this project.

References

1. M.P.Allen, D.J.Tildesley, Computer Simulation of Liquids

2. M.S.Warren, J.K.Salmon, A portable parallel particle program, Computer Physics Communications 87 (1995) 266-290

3. P.Gibbon, Introducing PEPC-a parallel electrostatic plasma coulomb-solver, in Proc. DPG Spring Meeting Aachen March 24-28 (2003)

Appendix

Fortran-90 form

```
do z=1,npart
p=0
  do i=-1,1
    do j=-1,1
       c(i)=0
       d(j)=0
       do lam=1,nlev-1
         proti=1
         protj=1
         do g=1,lam
            proti=proti*(((1-i)/2)+sign(1,i)*ibits(pkey(z),2*g-2,1))
            protj=protj*(((1-j)/2)+sign(1,j)*ibits(pkey(z),2*g-1,1))
         end do
         c(i)= c(i) + (2**(2*lam-1))*proti
         d(j)= d(j) + (2**(2*lam-1))*protj
       end do
       c(i)=abs(i)*sign(1,i)*(c(i)+1)
       d(j)=abs(j)*sign(2,j)*(d(j)+1)
       K(i,j)=pkey(z)+c(i)+d(j)
       p_key = K(i,j)
do w=1,nei(p_key)
 di=sqrt(((x(box_list(p_key,w))-x(z))**2)+((y(box_list(p_key,w))-y(z))**2))
 if ((di.LE.radius).AND.(di.GT.0)) then
   p=p+1
   list_neigh(z,p) = box_list(p_key,w)
 else
   p=p+1
   list_neigh(z,p) = 0
 end if
end do
    end do
  end do
end do
```

Figures

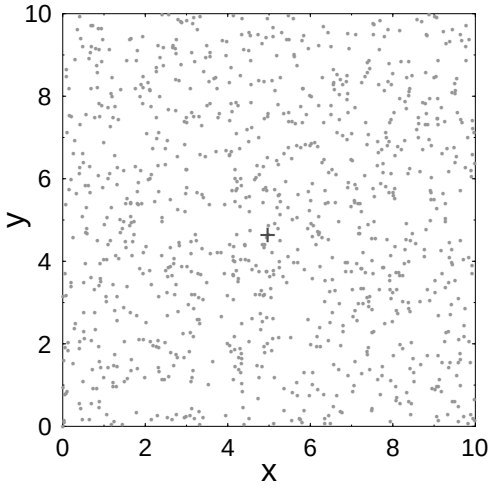


Figure 6: 1000 particles.

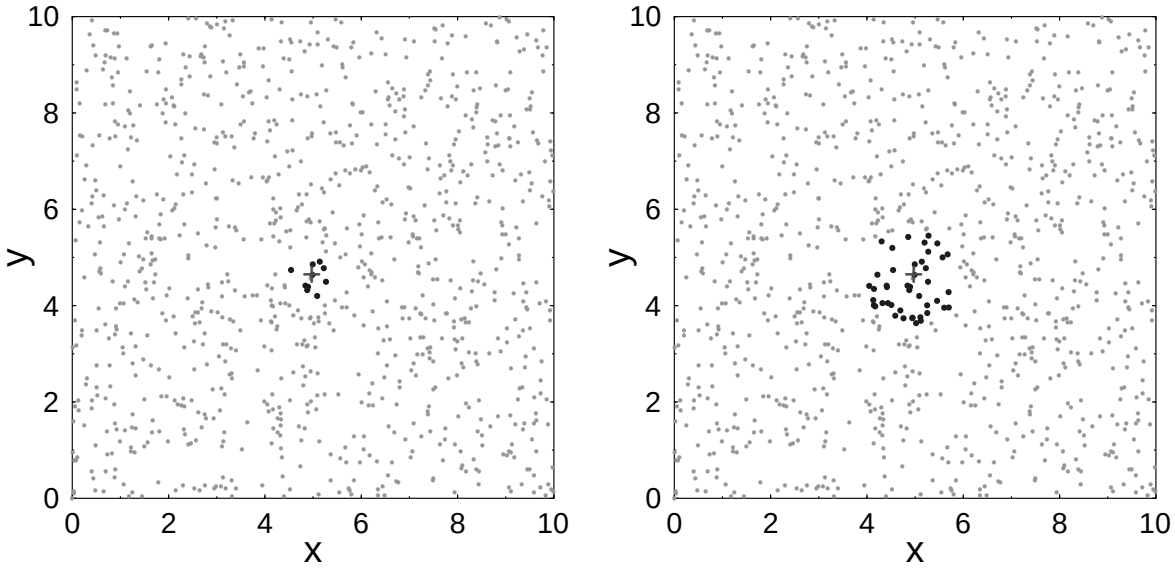


Figure 7: 5% and 10% of the box length.

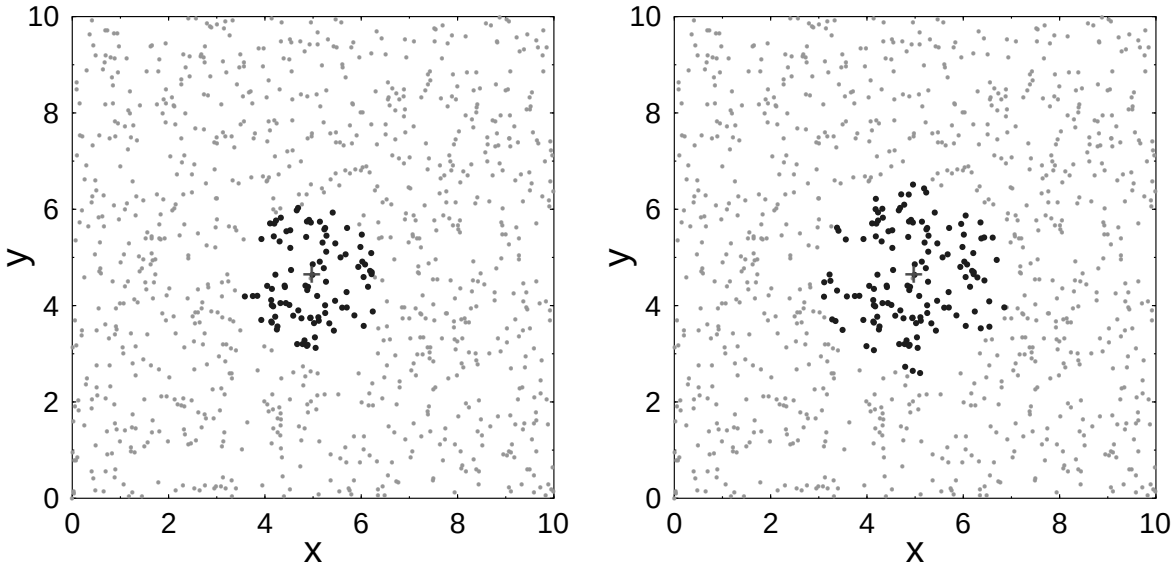


Figure 8: 15% and 20% of the box length.

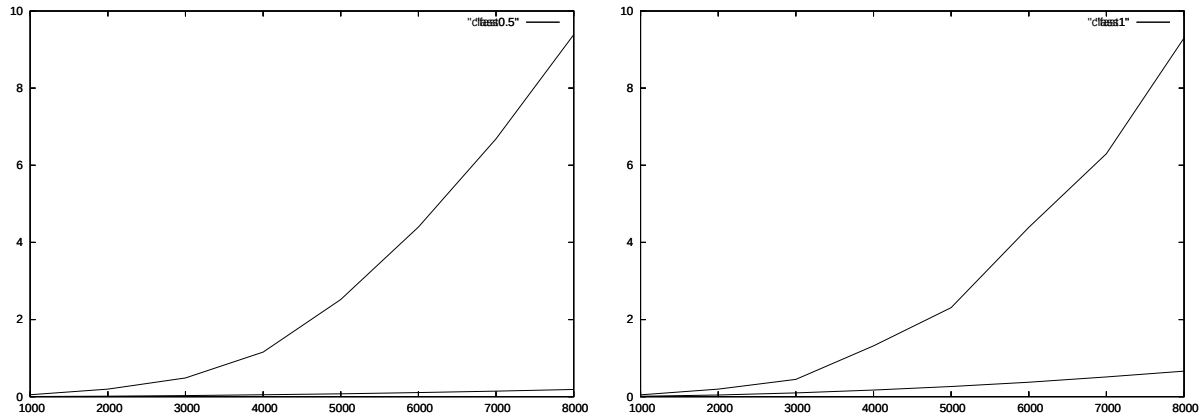


Figure 9: Time scalings for 5% and 10% of the box length.

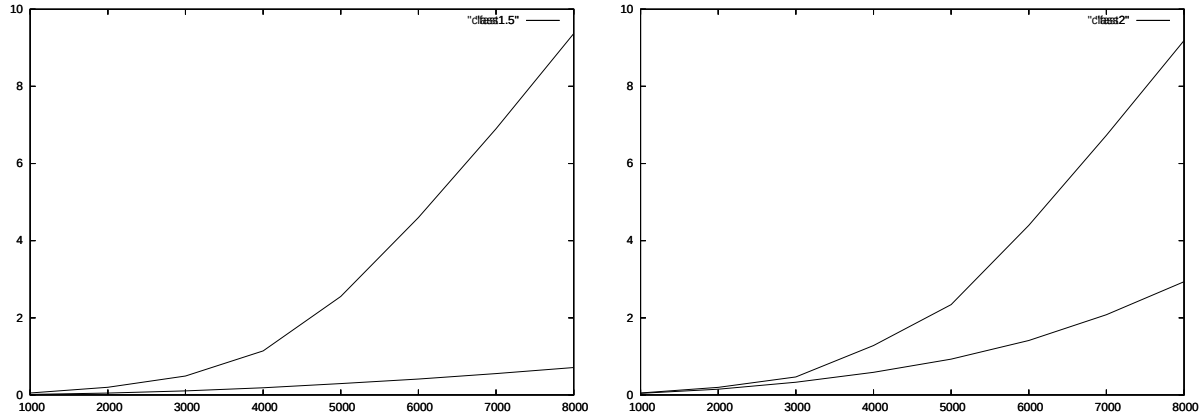


Figure 10: Time scalings for 15% and 20% of the box length.

The Estimation of Charge Extensions in the Continuous Fast Multipole Method (CFMM)

Ivo Kabadshow

Chemnitz University of Technology

E-mail: ivo.kabadshow@e-technik.tu-chemnitz.de

Abstract: The Fast Multipole Method is advantageous, if one wants to examine large numbers of particles because of its $\mathcal{O}(N)$ scaling. However, generalizing this method to continuous charges, correspondingly called Continuous Fast Multipole method, leads to new problems. Especially the treatment of the arising non-zero extents must be studied carefully. For s-type distributions one can calculate the extent analytically. However, such an analytical solution does not exist for higher angular momenta. For that reason one has to estimate an upper bound for these extents that reflects the real extent very well since it affects the performance of the Continuous Fast Multipole Method.

In computational physics one often deals with large ensembles of particles and their pairwise interactions. This is known as a N-body problem. Especially in present molecular dynamic calculations one has to evaluate the pairwise interactions, known as Coulomb interaction.

$$E_c = \sum_{i=1}^{N-1} \sum_{j=i+1}^{N} \frac{q_i q_j}{r_{ij}} \tag{1}$$

This equation sums up the Coulomb energy for N particles, with center $\mathbf{r}_i$ and charge q_i. However, for a large ensemble several problems arise from this equation. Because one had to take every interaction into account the computational cost scales $\mathcal{O}(N^2)$. For millions or even billions of particles the direct calculation of this potential is not reasonable because of large computational times. Also, it is not a helpful approximation to introduce a cut-off to the system and neglect the interaction between particles far from each other, because the coulomb potential is a long-range potential. To overcome the $\mathcal{O}(N^2)$ scaling another approach should be used. One method is the Fast Multipole Method introduced by Greengard [1]. The basic concept is to group distant particles and treat them like a single charge. If we use this approach, we can achieve $\mathcal{O}(N)$ scaling.

Fundamentals

Consider two charges located at positions $\mathbf{r}_2 = (r_2, \alpha, \beta)$ and $\mathbf{r}_1 = (r_1, \theta, \phi)$. The inverse distance between these two charges can also be written as an expansion of the associated Legendre polynomials P_{lm} under the condition $r_2 < r_1$. The bottom line is that $1/|\mathbf{r}_1 - \mathbf{r}_2|$ factorizes.

$$\frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} = \sum_{l=0}^{\infty} P_l(cos(\gamma)) \frac{r_2^l}{r_1^{l+1}} \qquad r_2 < r_1 \tag{2}$$

$$\frac{1}{|\mathbf{r}_1 - \mathbf{r}_2|} = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \frac{l-|m|}{l+|m|} \frac{r_2^l}{r_1^{l+1}} P_{lm}(cos(\alpha)) P_{lm}(cos(\theta)) exp(-im(\beta - \theta)) \tag{3}$$

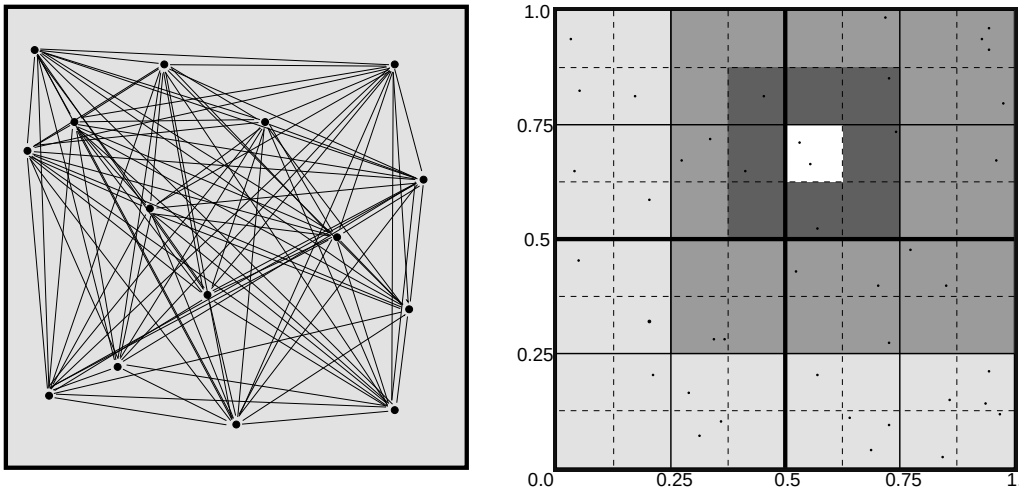


Figure 1: left: direct interaction; each line labels an interaction between two particles, since we have (e.g.) 15 particles there occur 105 interactions, right: scaled FMM box, all light-gray boxes can interact via multipoles with the white box.

Therewith one can define moments of a multipole expansion and coefficients of a Taylor expansion. This implicitly defines O_{lm} and M_{lm} as well.

$$\omega_{lm} = qO_{lm} = qa^l \frac{1}{(l+|m|)} P_{lm}(cos(\alpha))e^{-im\beta} \tag{4}$$

$$\mu_{lm} = qM_{lm} = q\frac{1}{r^{l+1}}(l-|m|)P_{lm}(cos(\theta))e^{im\phi} \tag{5}$$

Using the last three equations one can write the inverse distance as:

$$\frac{1}{|\mathbf{r_1} - \mathbf{r_2}|} = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} \omega_{lm}\mu_{lm} \tag{6}$$

The infinite sum in Eq. [6] can be approximated to any given precision by a finite sum.

FMM Translation Operators

The FMM needs three operators (A, B, C) to perform transformations on the multipole and Taylor expansions. If we want to shift a multipole expansion located at point $\mathbf{a}$ to point $\mathbf{a} + \mathbf{b}$ we use operator A.

$$\omega_{lm}(\mathbf{a} + \mathbf{b}) = \sum_{j=0}^{l} \sum_{k=-j}^{j} A_{jk}^{lm}(\mathbf{b})\omega_{jk}(\mathbf{a}) \tag{7}$$

Operator A is defined as:

$$A_{jk}^{lm}(\mathbf{b}) = O_{l-j,m-k}(\mathbf{b}) \tag{8}$$

The B operator is used to transform a multipole expansion centered at the origin into a local Taylor expansion about another center with shift $\mathbf{b}$

$$\mu_{lm}(\mathbf{a} - \mathbf{b}) = \sum_{j=0}^{\infty} \sum_{k=-j}^{j} B_{jk}(\mathbf{b})^{lm}\omega_{jk}(\mathbf{a}) \tag{9}$$

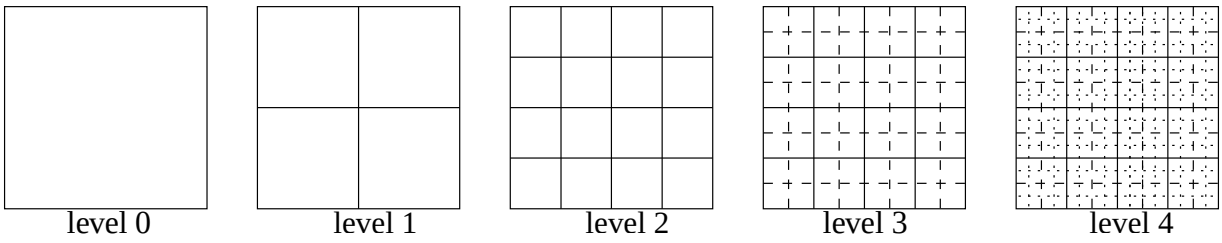


Figure 2: 2D FMM, every parentbox divides into 4 child boxes, level 4 contains 256 child boxes.

Operator B is defined as:

$$B_{jk}^{lm}(\mathbf{b}) = M_{j+l,k+m}(\mathbf{b}) \tag{10}$$

The third operator is required for translating a Taylor expansion at a point $\mathbf{r}$ about the origin to a Taylor expansion about point $\mathbf{a}$

$$\mu_{lm}(\mathbf{r}-\mathbf{a}) = \sum_{j=0}^{\infty}\sum_{k=-j}^{j} C_{jk}^{lm}(\mathbf{a})\mu_{jk}(\mathbf{r}) \tag{11}$$

Operator C is defined as:

$$C_{jk}^{lm}(\mathbf{a}) = A_{lm}^{jk}(\mathbf{a}) = O_{j-l,k-m}(\mathbf{a}) \tag{12}$$

The Fast Multipole Method

Building up the FMM tree - initial step

Every particle is represented by its charge q_i and its coordinates x_i, y_i, z_i. First we have to scale the particle coordinates to fit into a box with coordinate range $[0..1, 0..1, 0..1]$. This parent box is divided into a set of 8 equal child boxes. Each child box is subdivided recursively to build up the tree. The depth of the tree, respectively the number of divisions, is chosen to keep the number of particles in the lowest level box approximately independent from the total number of particles in the simulation box. A depth d leads to 8^d child boxes.

Generating and Translating multipoles - Pass 1

Having sorted all particle coordinates into the lowest level child boxes one can build up a multipole expansion in each lowest level box about its center. Now we can shift the multipole expansion of each lowest level box to the center of the associated parent box using operator A. These translated moments are summed and stored in the parent box. The procedure is continued until we reach level 3. This down-top shifting makes all children multipole expansions available in their parent boxes.

Converting Multipole Expansions into Taylor Coefficients - Pass 2

The generated box structure is used to determine whether particles (or boxes) can interact via multipole expansions or not. Interaction is only possible if the considered boxes are "well separated". Thus all particles in the first box must be well separated from all particles in the second box. The most distant particle in a box can be positioned in one of eight box corners. Therefore one has to draw a sphere around the box center with radius $\sqrt{3}d$. Such as Fig. [3] shows $2d$ is defined as the box length. Since the sphere overlaps with neighboring boxes, only interaction between particles in boxes which are not nearest

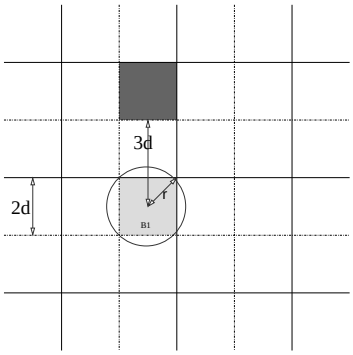


Figure 3: The sketch shows the minimal distance for two interacting boxes. The shaded box is well-separated from the box B_1, thus $ws = 1$. The length of each box is chosen to $2d$. The most distant charges in the corners of box B_1 have a distance of $\sqrt{3}d$ to the center.

neighbors are allowed. Now we can convert the distant multipole expansions into Taylor expansions about the center of box B_1. Therewith we create a local expansion representing all distant boxes. To achieve linear scaling, we only transform multipole expansions from boxes at the same level which are well separated from B_1, but are not well separated from B_1's parent box. The remaining child boxes interact via their parent boxes. To take the neglected information into account, we need the third operator C.

Local Taylor Expansions of Full Far-Field Potential - Pass 3

The third part is responsible for the transfer of neglected information from a parent box to a child box. It translates the parent's Taylor expansion to the center of all parent's children. This top-down shifting is repeated until we have reached the lowest level boxes of the FMM tree. Now all boxes contain the Taylor expansion from all "well-separated" boxes. Pass 3 is the converse of pass 1, where we shift the multipole expansions up the tree.

Particle Far-Field Interaction - Pass 4

This pass calculates the far-field potential. Each lowest level box contains a Taylor expansion representing all "well-separated" particles. By summing up the box energies $\sum_{lm} \omega_{lm}\mu_{lm}$ we get the far field energy.

Particle Near-Field Interaction - Pass 5

The remaining near field interaction of particles not "well separated" is done in the fifth pass. All remaining neighboring particles which did not contribute their charge to the current box interact directly. Now all interaction had been carried out and the total potential can be calculated by summing up the far and near field parts.

Parameters of the FMM are the length of the multipole expansion, the "well-separated" index ws and the depth of the FMM tree.

Gaussian Basis Sets

So far we can only treat point charges. But ab-initio calculations on Hartree-Fock or DFT level deal with charge distributions instead of point charges. Within the next section we want to clarify the use of Gaussian basis sets. Historically, the ab-initio calculations for molecules were performed with LCAO, i.e. Linear Combination of Atomic Orbitals. Thus, molecular orbitals are formed as a linear combination of atomic orbitals:

$$\psi_j = \sum_{i=1}^{n} c_{ij}\phi_i \tag{13}$$

where ψ_j is the j-th molecular orbital, c_{ij} are the coefficients of linear combinations, ϕ_i is the i-th atomic orbital, and n is the number of atomic orbitals. Solving the Hartree-Fock equations one obtains Molecular Orbitals (MO), describing the wavefunction for a single electron. Some implementations use Slater-Type Orbitals (STO's) due to their similarity to atomic orbitals of the hydrogen atom. With spherical coordinates one can describe them as follows:

$$\phi_i(\alpha, n, l, m, r, \theta\, \phi) = N r^{n-1} e^{-\alpha r} Y_{lm}(\theta, \phi) \tag{14}$$

where N is normalization constant, α is the Slater exponent, r, θ, ϕ are spherical coordinates, and Y_{lm} is the angular momentum part and is described by spherical harmonics. The values n,m and l represent the principal, angular momentum and magnetic quantum numbers. Unfortunately, functions of this kind are not suitable for calculations of two-electron integrals needed by the CFMM. For this reason Gaussian Type Orbitals are used. By summing up a certain number of GTO's with different exponents and coefficients one can reproduce the shape of the original STO, however with a much easier handling with respect to integration. Even with ten contracted Gaussians the two electron integral can be solved faster than by the original method (Fig. [4]). The Gaussian type orbitals (GTO's) can be written as:

$$G_{lmn}(x, y, z) = N e^{-\alpha r^2} x^l y^m z^n \tag{15}$$

where N is a normalization constant, α is called "Gaussian exponent". The x,y,z are Cartesian coordinates. Notice that in this formula, l, m and n are not quantum numbers but integral exponents. These basis functions or primitives can approximate s, p, d, f orbitals. Possible Cartesian Gaussian functions are shown in Fig. [4]

The sum of the exponents at Cartesian coordinates $(l + m + n)$ is used to mark functions as s-type ($l + m + n = 0$), p-type ($l + m + n = 1$), d-type ($l + m + n = 2$), etc.

The Gaussian Product Theorem

The Gaussian Product Theorem shows that a product of two arbitrary angular momentum Gaussian functions can be written as a Gaussian function again:

$$\begin{aligned} G_1 G_2 &= G_1(\alpha_1, \mathbf{A}, l_1, m_1, n_1) G_2(\alpha_2, \mathbf{B}, l_2, m_2, n_2) \\ &= exp\left[-\alpha_1\alpha_2(\overline{\mathbf{AB}})^2/\gamma\right] \\ &\times \left[\sum_{i=0}^{l_1+l_2} f_i(l_1, l_2, \overline{\mathbf{PA}}_\mathbf{x}, \overline{\mathbf{PB}}_\mathbf{x}) x_P^i e^{-\gamma x_p^2}\right] \\ &\times \left[\sum_{j=0}^{m_1+m_2} f_j(m_1, m_2, \overline{\mathbf{PA}}_\mathbf{y}, \overline{\mathbf{PB}}_\mathbf{y}) y_P^i e^{-\gamma y_p^2}\right] \\ &\times \left[\sum_{k=0}^{n_1+n_2} f_k(n_1, n_2, \overline{\mathbf{PA}}_\mathbf{z}, \overline{\mathbf{PB}}_\mathbf{z}) z_P^i e^{-\gamma z_p^2}\right] \end{aligned} \tag{16}$$

$$
\begin{aligned}
s &= Ne^{-\alpha r^2} \\
p_x &= Ne^{-\alpha r^2} x \\
p_y &= Ne^{-\alpha r^2} y \\
p_z &= Ne^{-\alpha r^2} z \\
d_{xx} &= Ne^{-\alpha r^2} x^2 \\
d_{xy} &= Ne^{-\alpha r^2} xy \\
d_{xz} &= Ne^{-\alpha r^2} xz \\
d_{yy} &= Ne^{-\alpha r^2} y^2 \\
d_{yz} &= Ne^{-\alpha r^2} yz \\
d_{zz} &= Ne^{-\alpha r^2} z^2
\end{aligned}
$$

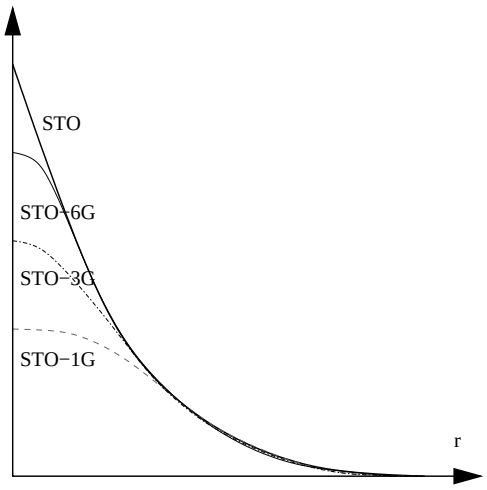


Figure 4: left: possible Cartesian Gaussian functions, right: a Slater-Type orbital (STO) is approximated by several Gaussian orbitals forming a contraction. STO-1G contains a single gaussian, STO-3G contains 3 gaussians, etc.

The charge distribution $\chi_1\chi_2$, a product of two gaussian basis functions can be expressed in cartesian or spherical coordinates:

$$
\begin{aligned}
\chi_1\chi_2 &= N_1 N_2 x^{l_1+l_2} y^{m_1+m_2} z^{n_1+n_2} e^{-(\alpha_1+\alpha_2)r^2} && (17) \\
\chi_1\chi_2 &= N_1 N_2 r^{l_1+m_1+n_1+l_2+m_2+n_2} \sin^{l+m}(\theta)\cos^{n}(\theta)\cos^{l}(\phi)\sin^{m}(\phi) e^{-(\alpha_1+\alpha_2)r^2} && (18)
\end{aligned}
$$

If we neglect all angular-dependent parts the product $\chi_1\chi_2$ simplifies to

$$
\chi_{12} = N_\chi r^{l_1+m_1+n_1+l_2+m_2+n_2} exp\left[-(\alpha_1+\alpha_2)r^2\right] \tag{19}
$$

with a normalization constants N_1, N_2 is

$$
\begin{aligned}
N_1 &= \left(\frac{2\alpha_1}{\pi}\right)^{3/4} \frac{(2\sqrt{\alpha_1})^{l_1+m_1+n_1}}{\sqrt{(2l_1-1)!!(2m_1-1)!!(2n_1-1)!!}} \\
N_2 &= \left(\frac{2\alpha_2}{\pi}\right)^{3/4} \frac{(2\sqrt{\alpha_2})^{l_2+m_2+n_2}}{\sqrt{(2l_2-1)!!(2m_2-1)!!(2n_2-1)!!}}
\end{aligned}
$$

The Continuous Fast Multipole Method (CFMM)

Differences between FMM and CFMM

The classical point charge FMM systematically organizes multipole representations of local charge distributions, so that each particle interacts with local expansions of the potential due to all distant particles. To group particles they are placed in a box which is repeatedly subdivided to create local collections of point charges. Local and distant distributions are distinguished by the global well-separated index (ws), defined as the number of boxes which must separate two collections of charges before they may be considered distant, and can interact through multipole expansions. However, this method fails for continuous distributions, because a single distribution can cover the whole box.

CFMM is a generalization of the Fast Multipole Method, which allows one to calculate the Coulomb interactions of a collection of finite extent distributions represented by continuous functions scaling linearly with system size. A linear scaling method based on the FMM approach rapidly handles the distant

distributions by collecting charge distributions and expanding them into multipoles. The remaining local contributions must be evaluated explicitly, but their number is only linear, scaling as $\mathcal{O}(NM)$ where M is an effective number of neighboring charges The CFMM takes account of the extent of distributions by assigning a "well-separated" index value to each distribution based on the following equation:

$$\text{ws}_{\text{CFMM}} = \max(2[r_{\text{ext}}/l], \text{ws}_{\text{ref}}) \tag{20}$$

where l is the length of a box, and ws_{ref} is the ws value chosen for point charges. Thus the ws index and hence the number of interacting neighbor boxes is determined by the extent of the distribution. Distributions covering the entire box, cannot interact via multipoles and have to be computed directly. In comparison to the FMM, where we have to sum up over all particles in a box to gain the multipole moment, CFMM requires integration over charge distributions.

$$\text{FMM:} \quad \omega_{lm} = \sum_j q_j r_j^l \frac{P_{lm}(sin(\theta_j), cos(\theta_j))}{(l+m)!} (cos(m\phi_j) - i \cdot sin(m\phi_j)) \tag{21}$$

$$\text{CFMM:} \quad \omega_{lm} = \int_0^\infty \int_0^\pi \int_0^{2\pi} \chi_i \chi_j r^l \frac{P_{lm}(sin(\theta), cos(\theta))}{(l+m)!} \tag{22}$$

$$\cdot (cos(m\phi_j) - i \cdot sin(m\phi_j))\, r^2 sin(\theta) d\phi d\theta dr \tag{23}$$

The restriction to point charges means the FMM is not immediately applicable to problems in which the charge distributions have significant extent. But within a given error bound it is possible to treat charge distributions as point charges.

Computing the two-Electron Integral

A charge distribution is composed of two Gaussian basis functions. For s-type basis functions ($l + m + n = lmn = 0$) one can solve the two electron integral analytically. For a given s-type Gaussian

$$\chi_s = \frac{2\alpha}{\pi}^{\frac{3}{4}} e^{-\alpha r^2} \tag{24}$$

the integral with four s-type functions (two distributions) can be written as:

$$\langle \chi_{s1}\chi_{s2} | \chi_{s3}\chi_{s4} \rangle = \left(\frac{2\alpha_1}{\pi}\right)^{\frac{3}{4}} \left(\frac{2\alpha_2}{\pi}\right)^{\frac{3}{4}} \left(\frac{2\alpha_3}{\pi}\right)^{\frac{3}{4}} \left(\frac{2\alpha_4}{\pi}\right)^{\frac{3}{4}} \int\int \frac{e^{-\gamma \mathbf{r}_1^2} e^{-\delta |\mathbf{r}_2 - R|^2}}{|\mathbf{r}_1 - \mathbf{r}_2|} d\mathbf{r}_1 d\mathbf{r}_2 \tag{25}$$

in which α_1 and α_2 are the Gaussian exponents of the first charge distribution, α_3 and α_4 the exponents of the second charge distribution, γ is the sum of α_1 and α_2, and δ the sum of α_3 and α_4. R is defined as the distance between these two distributions.

$$\langle \chi_{s1}\chi_{s2} | \chi_{s3}\chi_{s4} \rangle = \frac{\text{erf}\left(\sqrt{\frac{\gamma\delta}{\gamma+\delta}} R\right)}{R} \tag{26}$$

$\text{erf}(x)$ is the error-function and defined as:

$$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_{-\infty}^{x} e^{-t^2} dt \tag{27}$$

For large R, increasing separation respectively, the two charge distributions can be treated as classical point charges.Thus we can define an error:

$$\epsilon(\gamma, \delta, R) = \frac{1}{R} - \frac{\text{erf}\left(\sqrt{\frac{\gamma\delta}{\gamma+\delta}} R\right)}{R} \tag{28}$$

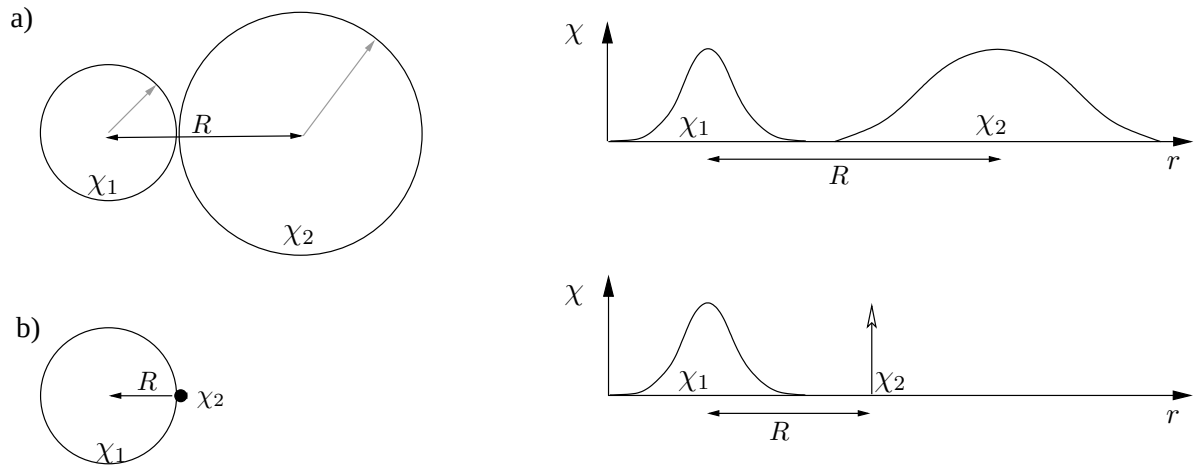


Figure 5: Gaussian charge distribution: a) two distributions having non-zero extents, every distribution contributes to R, b) one charge distribution and one point charge, only the charge distribution contributes to R

Using the relation $\text{erfc}(x) = 1 - \text{erf}(x)$ we can write:

$$\epsilon(\gamma, \delta, R) = \frac{\text{erfc}\left(\sqrt{\frac{\gamma\delta}{\gamma+\delta}}R\right)}{R} \tag{29}$$

By defining one of these charge distribution as a point charge, the two-electron integral Eq. [25] can be used to calculate the extent of the remaining charge distribution. Therefore two of four Gaussian exponents are chosen infinite. Treating for example the second distribution as a point charge with $\alpha_3 \to \infty$, $\alpha_4 \to \infty$ Eq. [30] gives us the extent of the first charge distribution.

$$\epsilon(\gamma, R_\gamma) = \frac{\text{erfc}\left(\sqrt{\gamma}R_\gamma\right)}{R_\gamma} \tag{30}$$

For higher angular momenta distributions the two electron integral cannot be solved analytically. For this reason we have to estimate an upper limit for the extent of these distributions. Since we can solve the two electron integral analytically for s-type charge distributions, it would be helpful to use these s-type distributions to estimate the extent for higher angular momenta functions. As a first step we tried to approximate the basis functions by s-type basis functions. A given arbitrary basis function

$$\chi = \left(\frac{2\alpha}{\pi}\right)^{\frac{3}{4}} \frac{(2\sqrt{\alpha})^{l+m+n}}{\sqrt{(2l-1)!!(2m-1)!!(2n-1)!!}} r^{l+m+n} e^{-\alpha r^2} \tag{31}$$

is estimated by a s-type basis function

$$\chi_s = c\left(\frac{2\beta}{\pi}\right)^{\frac{3}{4}} e^{-\beta r^2} \tag{32}$$

Satisfying the following conditions, we find a optimal s-type basis function for a given arbitrary type of basis function with the property $\chi_s > \chi$

$$\chi = \chi_s \tag{33}$$

$$\frac{d\chi}{dr} = \frac{d\chi_s}{dr} \tag{34}$$

$$\frac{d^2\chi}{dr^2} = 0 \tag{35}$$

These equations can be solved analytically and one gets:

$$\beta = \frac{\alpha(\sqrt{8(lmn)+1}+1)}{2(lmn)+\sqrt{8(lmn)+1}} \tag{36}$$

$$c = \frac{(2(lmn)+1+\sqrt{8(lmn)+1})e^{-1/2(lmn)}}{\left[\frac{\sqrt{8(lmn)+1}+1}{2(lmn)+1+\sqrt{8(lmn)+1}}\right]^{3/4}\sqrt{(2l-1)!!(2m-1)!!(2n-1)!!}} \tag{37}$$

where (lmn) is the shortened syntax for $l+m+n$ We can now estimate the extent of higher angular momenta basis functions.

α	$l+m+n$	β	c
1	1	0.666666	2.013709
2	1	1.333333	2.013709
3	1	1.999999	2.013709
1	2	0.561553	2.987065
2	2	1.123105	2.987065
3	2	1.684658	2.987065

Table 1: calculated parameters β and c for a given α and lmn, a small β denotes a large R and vice versa.

Since we are treating charge distributions and not only basis functions, the resulting β is not necessarily the best solution. Thus, it would be better to estimate the extent of a product of basis functions, compared to only one basis function. This idea leads us to the following equation for arbitrary angular momenta distributions:

$$\chi_1\chi_2 = N_1N_2r^{(lmn)_1+(lmn)_2}e^{-(\alpha_1+\alpha_2)r^2} \tag{38}$$

Our s-type charge distribution now looks like:

$$\chi_{1s}\chi_{2s} = c\left(\frac{2\beta_1}{\pi}\right)^{3/4}\left(\frac{2\beta_2}{\pi}\right)^{3/4}e^{\left[-(\beta_1+\beta_2)r^2\right]} \tag{39}$$

The conditions look similar to the basis function estimate.

$$\chi_1\chi_2 = \chi_{s1}\chi_{s2} \tag{40}$$

$$\frac{d(\chi_1\chi_2)}{dr} = \frac{d(\chi_{s1}\chi_{s2})}{dr} \tag{41}$$

$$\frac{dR}{d\beta_1} = 0 \tag{42}$$

$$\frac{dR}{d\beta_2} = 0 \tag{43}$$

At this point we want to emphasize that β_1 and β_2 must have the same value to obtain the minimal R. So we can simplify a little and set $\beta_1 = \beta_2 = \beta$. The final equation can be written as:

$$\frac{\text{erfc}(\sqrt{2\beta}R(\beta))}{R(\beta)} = \frac{1}{N_\chi}\left(\frac{2}{\pi}\right)^{3/2}\left(\frac{2e}{n}\right)^{n/2}\varepsilon\beta^{3/2}\left[(\alpha_1+\alpha_2)-2\beta\right]^{n/2} \tag{44}$$

where R is the extent of the charge distribution, N_χ the normalization constant and ϵ the given error bound. Unfortunately there exists no analytical solution for R and β, so one has to find the solution numerically. To simplify the expression we can write:

$$\frac{\text{erfc}(\sqrt{2\beta}R)}{R} = g(\beta) \tag{45}$$

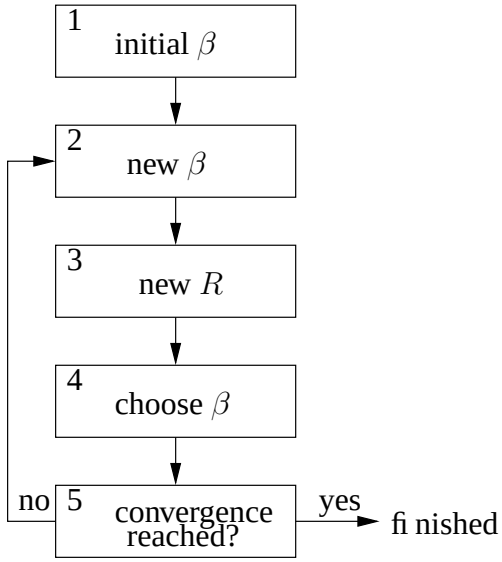


Figure 6: computational scheme finding β and R

R implicitly depends on β. However, using the conditions $\frac{dR}{d\beta_1} = 0$ and $\frac{dR}{d\beta_2} = 0$ leads to

$$R = \sqrt{\frac{\ln\left(\frac{\sqrt{2}}{\sqrt{\pi\beta}(-\frac{dg}{d\beta})}\right)}{2\beta}} \tag{46}$$

Unfortunately the attained $R(\beta)$ dependency is not really helpful. If one puts this equation into the final equation [44] to eliminate the R dependency, one has to deal with a lot of discontinuity problems instead, especially for higher angular momenta. Several attempts to solve the resulting equation with a simple newton algorithm failed because of these discontinuities. For that reason we chose another approach.

Another approach for further decreasing R

Considering Eq. [45] again:

$$\frac{\text{erfc}(\sqrt{2\beta}R)}{R} = g(\beta) \tag{47}$$

We can implicitly plot the $R = R(\beta)$ dependency (Fig. [7]). From this equation we can also derive the limits β_1, β_2 in-between the minimal R can be found:

$$\text{lower limit:} \quad \beta_{i1} = \frac{3}{2}\frac{\alpha_1 + \alpha_2}{n+3} \tag{48}$$

$$\text{upper limit:} \quad \beta_{i2} = \frac{1}{2}(\alpha_1 + \alpha_2) \qquad \beta_{i1} < \beta < \beta_{i2} \tag{49}$$

Performing a binary search on the given interval $[\beta_1, \beta_2]$, we can find a β and a corresponding minimal R within a given precision. However, binary search converges too slowly. Treating millions or billions of distributions possibly slows down the entire calculation, since we have to calculate an extent for every charge distribution.To improve the performance the binary search is replaced by a Fibonacci search that scales only $\mathcal{O}(log(n))$.

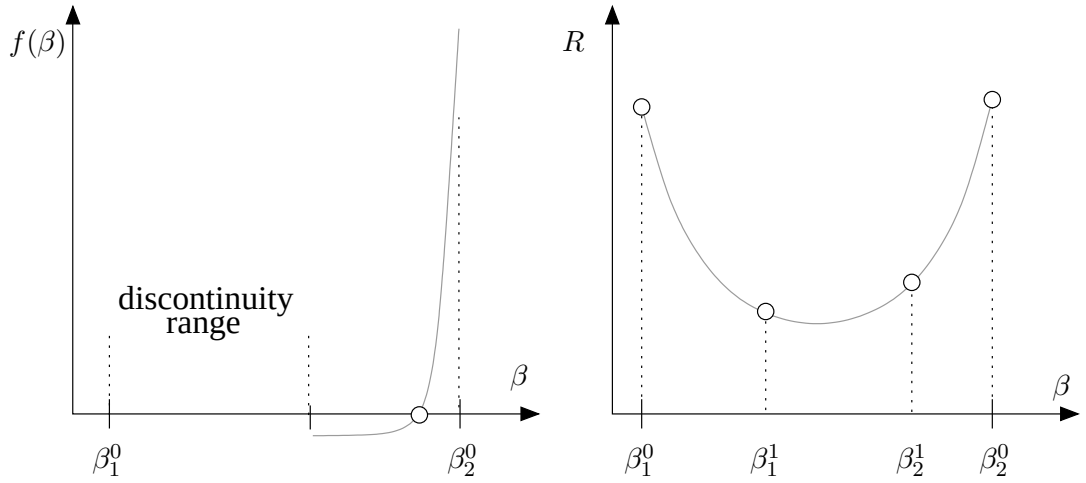


Figure 7: left:using condition $dR/d\beta = 0$ our function f depends on β only, however discontinuities occur. right: implicit plot $R(\beta)$. Between the considered limits this function is unimodal, and thus the minimum $dR/d\beta = 0$ can be determined by a Fibonacci search.

Fibonacci Line Search

The Fibonacci Line Search algorithm allows to find a maximum of a unimodal function $f(x)$ over a given interval $[\beta_{i1}, \beta_{i2}]$. A user-defined precision allows to limit the total number of iterations. This limit is defined as:

$$F_n > (\beta_{i2} - \beta_{i1})/eps \tag{50}$$

where F_n is the n-th Fibonacci number and β_{i1}, β_{i2} the given search interval. The Fibonacci numbers are defined in the following manner:

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \quad \text{for} \quad n \geq 2 \end{aligned}$$

The search concept involves placing two test points in-between the interval limits $[\beta_{i1}, \beta_{i2}]$ using the ratio of Fibonacci numbers. The test points are positioned at:

$$\beta_{i1}^1 = \beta_{i1}^0 + \frac{F_{n-2}}{F_n}(\beta_{i2}^0 - \beta_{i1}^0) \tag{51}$$

$$\beta_{i2}^1 = \beta_{i2}^0 + \frac{F_{n-1}}{F_n}(\beta_{i2}^0 - \beta_{i1}^0) \tag{52}$$

Now the function is evaluated at these two new points. Since we want to minimize our function, we keep the smaller functional value and the opposite interval-limit. After all required iterations have been passed through, a final interval is the answer. The procedure is also illustrated on Fig. [6].

Determining R by a Newton algorithm.

To choose the new interval limits we need the functional values $R(\beta_{i1}^n)$ and $R(\beta_{i2}^n)$. These are determined by a Newton algorithm.

$$f(R) = \text{erfc}(\sqrt{2\beta}R) - g(\beta)R \tag{53}$$

$$R_{n+1} = R_n - \frac{f(R_n)}{f'(R_n)} \tag{54}$$

The initial value for this iterative Newton algorithm is $R_0 = 0$.

Comparison of the determined extents

Obviously the second algorithm finds a lower upper limit for the extent of higher angular momenta distributions. Especially for large distributions with small β the latter method allows minimizing the costs for a direct interaction of these charge distributions. (Tab. [2])

α	$l+m+n$	β 1st algorithm	R_1 1st algorithm	β 2nd algorithm	R_2 2nd algorithm
1	1	0.666666	0.608559	0.821384	0.570837
2	1	1.333333	0.260212	1.579524	0.247564
3	1	1.999999	0.144235	2.327489	0.138089
1	2	0.561553	0.649036	0.696932	0.618961
2	2	1.123105	0.221573	1.279649	0.216687
3	2	1.684658	0.100091	1.859443	0.098656

Table 2: obtained extents from approximated basis functions and approximated charge distributions. The estimated upper bound can be reduced by using the second algorithm.

Conclusions

We have introduced an algorithm to perform extent estimations of high angular momenta charge distributions. It was shown that the upper bound for these distribution extents can be reduced prefering a combined Fibonacci-Newton algorithm on the charge distribution instead of using a Newton algorithm on a single basis function.

Acknowledgements

This work was carried out in the framework of the ZAM guest student program at the research centre Juelich. I want to thank everybody supporting this program. Special thanks go to Dr. R. Esser for the coordination of the program and my supervisor Dr. H. Dachsel for his encouragement to circumnavigate the appearing numerical cliffs.

References

1. L. Greengard, V. Rokhlin J. Comput. Phys. 60 (1885) 187.
2. C. A. White, B. G. Johnson, P. M. W. Gill, M. Head-Gordon Chem.Phys.Letters 253 (1996) 268.
3. C. A. White, M. Head-Gordon J. Chem. Phys. 101 (1994) 6593
4. C. A. White, M. Head-Gordon J. Chem. Phys. 105 (1996) 5061
5. L. Greengard, V. Rokhlin J. Comp. Phys. 135 (1997) 280
6. C. A. White, B. G. Johnson, P. M. W. Gill, M. Head-Gordon Chem.Phys.Letters 230 (1994) 8.

Quantum Chemical Investigations on the Auger Process Induced Fragmentation of Organic Compounds and its Signifi cance to DNA Irradiation Damage

Slawomir Pitula, Jonas Wiebke

University of Cologne
Department of Chemistry

s.pitula@fz-juelich.de
mail@jonaswiebke.de

Abstract:

^{125}I transforms to highly excited ^{125}Te via electron capture [1], leaving core shell vacancies that result in massive Auger emission. Molecular systems incorporating ^{125}I will, following the ^{125}I decay, fragment into molecular and atomic positive ions with individual charges of up to +18 [2], a process commonly referred to as "Coulomb explosion". Auger emitters are highly radiotoxic, and have been subject to radiomedical research for decades [3]. Whereas the investigation of the effects Auger processes on biological systems could improve the understanding of basic aspects of radiotoxicity in general, ^{125}I Auger electron emission may play a pioneering role in therapeutical applications, such as tumor-specifi c radionuclide deposition.

However, there is still much uncertainty in the molecular and cellular mechanisms following Auger processes *in vivo*. In this work, the response of a ^{77}Br-substituted 10 base pair DNA sequence on excessive ionization was investigated by quantum chemical methods on a DFT level of theory. Here and in further systems ^{77}Br was chosen as the Auger emitter instead of ^{125}I, to overcome certain problems as discussed below. In a fi rst approach, the Coulomb explosion of $CH_3{}^{125}I$, $C_2H_5{}^{125}I$, and n-$C_3H_7{}^{125}I$, was studied systematically to draw connections between electronic structures and observed fragmentation patterns. This was done by considering the Auger electron emission as being fi nished, and calculating the nuclear relaxation for the electronic ground state of the molecular ions. More calculations were carried out on the Coulomb explosion of ^{125}Iodouracil, ^{77}Bromouridine-5′-monophosphate, and the ^{77}Bromouridine-5′-monophosphate−Adenosine-5′-monophosphate nucleotide pair.

Introduction

It was not until the early 1970s when Hofer et al. [4, 5], and Feindegen et al. [6, 7] confirmed chromatin damage and the impressive radiotoxic effects due to ^{125}I Auger electron emission in mammalian cells experimentally. Several pathways to cell death have so far been discussed, including radiation and low-energy electron bombardment due to the initial decay, ^{125}Te recoil energy and the chemistry involved with the transformation of ^{125}I to ^{125}Te, and Coulomb explosion of molecules covalently binding ^{125}I. The radiomedial aspects of Auger Processes appear to be a surprisingly wide field of research. Despite that, the chain of cause and effect, concerning Auger processes *in vivo*, is far from being revealed.

The cell lethality pattern due to intracellular Auger emission is comparable to that due to α particle exposure. It is confirmed that the effect of ^{125}I Auger emission on cellular structure critically depends on the intracellular location of the decay. ^{125}I located outside of the cell nucleus, e.g. plasma membrane-bound via ^{125}I-labeled Concanavalin A, is relatively non-toxic [8], requiring $\sim$ 20,000 decays per cell to cause 50% cell death, compared to $\sim$ 60 decays per cell for DNA incorporated ^{125}IUdR.

Considering intranuclear ^{125}I decay, there is evidence that there is no direct correlation between radiation or particle emission energy deposition and nucleus damaging. For example, ^{125}I-labeled oligonucleotides not inserting in DNA, but being accumulated in the nucleus by liposome transportation, only gave lethalities comparable to extranuclear Auger processes [9]. In contrast, DNA damage——which is closely connected to cell death——is mostly due to decay events of direct DNA-bound Auger emitters, causing Single (SSBs) and Double Strand Breaks (DSBs) in DNA, and, more globally, mutations and chromosome aberrations. ^{125}I-labeled oligonucleotides incorporated in DNA caused SSBs and DSBs within a range of 10 base pairs of the decay site [10]. DNA triplex forming ^{125}I-labeled oligonucletides yielded similar results, though located at a seperate DNA strand [11].

Stating that intranuclear ^{125}I decay alone was responsible for cell death is furthermore challenged by the findings of several groups that chemical radioprotectors as Dimethylsulfoxide [12], Vitamin C [13], and others, can mitigate the radiotoxic effects of Auger processes. Though they exhibit similar lethality patterns, this is not true for α radiation [12]. So, evidence arises that DNA damage due to Auger processes may be mediated by somewhat indirect effects [14], e.g. water radicals from DNA solvation shells, or reactive organic fragments due to Auger process induced Coulomb explosions.

Using calculated Auger electron emission spectra, e.g. by Monte Carlo methods [15], several theoretical models have been applied to investigate DNA [16] and nucleosome DNA superstructure [17] damage due to Auger processes. On a very qualitative level, these and other theoretical approaches fit to experimental observation, agreeing that indirect effects to DNA damage due to Auger processes cannot be neglected, if they are not even a major part.

Because of their striking radiotoxicity and, on the other hand, their strong dependence on intracellular location, several radiomedical applications of Auger emitters are recently discussed, reviewed e.g. by Hofer [3]. For example, in radiodiagnostics, one always has to find a reasonable compromise between detectable radioactivity, accumulated in a specific tissue type to investigate, and the least possible exposure of the system to damaging radiation effects, and Auger emitters could be applicable to that field. Second, the selective transport of Auger emitters into tumor cell nuclei, e.g. by tumor epitope-specific antibodies, could prove to deliver a great deal to cancer therapy and other types of molecular surgery.

The aim of this work is to find out if, and in which way bioorganic compounds will fragment due to the radioactive decay of the covalently bound Auger emitters ^{125}I and ^{77}Br, and which fragments will occur. To achieve comparable results, these questions were approached by calculations on the simple alkyliodides $CH_3{}^{125}I$, $C_2H_5{}^{125}I$, and n-$C_3H_7{}^{125}I$, because the Coulomb explosion $CH_3{}^{125}I$ and $C_2H_5{}^{125}I$ was investigated experimentally by mass spectrometry methods [2, 18, 19]. Further calculations focussed the bioorganic systems ^{125}Iodouracil (^{125}IUdR), ^{77}Bromouridine-5′-monophosphate, and the ^{77}Bromouridine-5′-monophosphate−Adenosine-5′-monophosphate nucleotide pair, because of their importance to radiomedical research.

Another intent was to prove if DFT is an applicable technique to calculate highly charged systems like products of Auger processes because most methods can hardly describe states which are not in their ground states. The investigation with DFT technique has of course several limitations and these were also to be detected by this work.

Theoretical Basics

Auger Process

The Auger process was first discovered in 1925 by the french physicist Pierre Auger [20] when he irradiated some atoms by low-energy x-rays and thereby removed a single electron from the core region of the atom. After the initial ionization process, the system was left in an highly excited electronic state,

possibly well above the ionization levels. It follows the fast electronic relaxation process to the various electronic ground states associated with a different number of electrons escaping from the system. The Auger process takes place on a time scale of 10^{-16} s $\leq t \leq 10^{-14}$ s [21], i.e. much faster than the relaxation process of the molecular geometry which appears on the timescale of a molecular vibration (10^{-12} s) [22].

Later it was proved that this Auger ionization process could be initiated by electron capture and ensuing nuclear conversion of the nucleus. Certain isotopes such as ^{125}I, ^{77}Br, known as Auger emitters, spontaneously convert into positively charged ^{125}Te and ^{77}Se. Molecules bound to Auger emitters turn into the corresponding positively charged molecular system with the Auger emitter replaced by its decay product at the molecular geometry of the initial compound.

Density Functional Theory

Density Functional Theory (DFT), which was developed by Thomas [23], Fermi [24], and Dirac [25] in the 1920s and 30s, became applicable for routine quantum chemical calculations in the early 1980s with the development of continuously more accurate exchange functionals. Because of its moderate scaling with molecular size, compared to HF, or more advanced approaches such as many-body-perturbation theory and coupled-cluster techniques, DFT has become a widely used tool for molecular structure calculations .

In DFT, the electron density $\rho(\mathbf{r})$, depending on the location $\mathbf{r}$, is chosen to be the basis variable, which is formally justified by the proof of the Hohenberg−Kohn theorems [26]

$$\rho_0(\mathbf{r}) = \sum_i \rho_i(\mathbf{r}) = \begin{cases} \sum_i^{\infty} n_i \int |\psi_i(\mathbf{x})|^2 d\sigma & \text{interacting, } 0 \leq n_i \leq 1 \\ \sum_i^{N} \int |\psi_i(\mathbf{x})|^2 d\sigma & \text{non-interacting} \end{cases} \tag{1}$$

where N is the number of electrons, and n_i are the occupation numbers (probabilities) of the spin orbitals $\psi_i(\mathbf{x}) = \psi_i(\mathbf{r}, \sigma)$. The integral is over the spin variable σ to give spin-independent densities $|\psi_i(\mathbf{x})|^2 = \rho_0(\mathbf{r})$.

The total electronic energy expression in terms of DFT, according to the Kohn−Sham method, is

$$E[\rho_0] = T[\rho_0] + U[\rho_0] + V[\rho_0] = F_{\mathrm{HK}}[\rho_0] + V[\rho] = T^{\mathrm{s}}[\rho_0] + J[\rho_0] + E_{\mathrm{XC}}[\rho_0] + V[\rho_0] \tag{2}$$

$$E_{\mathrm{XC}}[\rho_0] = T[\rho_0] - T^{\mathrm{s}}[\rho_0] + U[\rho_0] - J[\rho_0] \tag{3}$$

Here, $V[\rho_0]$ is the electron−nuclei potential functional given by the molecular geometry, and $E_{\mathrm{XC}}[\rho_0]$ is the exchange energy functional explained below. The universal Hohenberg−Kohn functional $F_{\mathrm{HK}}[\rho_0] = T[\rho_0] + U[\rho_0]$ includes several unknown non-classical contributions to the kinetic energy $T[\rho_0]$ and electron−electron interaction $U[\rho_0]$. However, this problem can be overcome indirectly by the Kohn−Sham method.

Its basic idea is to set up an non-interacting N electron reference system s as $\hat{H}^{\mathrm{s}} = -\sum_i \nabla_i^2/2 + \sum_i \hat{V}_i^{\mathrm{s}}(\mathbf{r})$ that gives the same (non-interacting) ground state density $\rho_0(\mathbf{r})$ as the (interacting) system to investigate does, where $\hat{V}_i^{\mathrm{s}}(\mathbf{r})$ is some general effective potential. Because $\hat{H}_{\mathbf{s}}$ is non-interacting, $\hat{H}^{\mathrm{s}}$ can be separated in a sum of N one-electron Hamiltonians $\sum_i \hat{h}_i^{\mathrm{s}} = \hat{H}^{\mathrm{s}}$, and the N eigenfunctions of $\hat{H}_{\mathbf{s}}$ can be expresses as a single Slater determinant $\Psi^{\mathrm{s}}(\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_N) = (N!)^{-1/2} \sum_n (-1)^n \hat{P}_n \prod_i \psi_i^{\mathrm{s}}(\mathbf{x})$. So, one has N one-electron problems instead of one N electron problem.

The Hohenberg−Kohn functional can then be handled in terms of this non-interacting reference system, that is $F_{\mathrm{HK}}[\rho_0] = T^{\mathrm{s}}[\rho_0] + J[\rho_0] + E_{\mathrm{XC}}[\rho_0] + V[\rho_0]$, where $T^{\mathrm{s}}[\rho_0]$ is the non-interacting kinetic energy contribution, and $J[\rho_0]$ the classical Coulomb repulsion. The unknown classical contributions are collected in the so called exchange energy functional $E_{\mathrm{XC}}[\rho_0]$, as given in (3). By the Kohn−Sham method,

one gets an indirect, but nevertheless exact description of an interacting system by a non-interacting model system, that does not exist physically, but gives the same energy and density.

One can express the energy (2) in terms of a set of N spin orbitals, $\{\psi_i^{\mathrm{s}}(\mathbf{x})\}$, by (1). Then, minimizing the energy functional $E[\rho_0] \longrightarrow E[\{\psi_i^{\mathrm{s}}(\mathbf{x})\}]$ with the N^2 constraints of spin orbital orthonormality, one obtains N one-electron equations of non-interacting type. Unitary transforming the orbitals $\{\psi_i^{\mathrm{s}}(\mathbf{x})\}$ then gives the canonical Kohn−Sham orbital equations

$$\varepsilon_i \psi_i^{\mathrm{s}}(\mathbf{x}) = \hat{h}_i^{\mathrm{s}} \psi_i^{\mathrm{s}}(\mathbf{x}) = \left(-\nabla^2/2 + \hat{W}(\mathbf{r})\right) \psi_i^{\mathrm{s}}(\mathbf{x}) = \left(-\nabla^2/2 + \hat{J}(\mathbf{r}) + \hat{E}_{\mathrm{XC}}(\mathbf{r}) + \hat{V}(\mathbf{r})\right) \psi_i^{\mathrm{s}}(\mathbf{x}) \quad (4)$$

Because of the dependence of $\hat{h}_i^{\mathrm{s}}$ on the $\{\psi_i^{\mathrm{s}}(\mathbf{x})\}$ via the Coulomb repulsion potential, the KS equations are non-linear, and have to be solved self-consistently. However—the KS equations have the same form as the canonical Hartree−Fock equations, except they consider a more general, but still local effective potential $\hat{W}(\mathbf{r}) \equiv \hat{V}^{\mathrm{s}}(\mathbf{r})$. In contrast to this, HF theory uses an essentially non-local exchange potential, and still is by definition approximate, considering no electron correlation at all.

Exchange Functionals

As one can see from the previous section, the exchange functional $E_{\mathrm{XC}}[\rho_0]$ proves to be the crucial part of any DFT calculation. Several approaches have been made to more and more accurate exchange functionals, including local spin density approach, LSDA, generalized gradient corrected approach, GGA, and hybride methods, mixing LSDA and GGA methods with HF exchange correlation. However, finding new and better exchange functionals still remains to be a challenging field of research.

In this work, the GGA type BP86 functional, and the hybride type functional B3LYP were used to cover different methods of exchange treatment. Both functionals, $E_{\mathrm{XC}}^{\mathrm{BP86}}$ and $E_{\mathrm{XC}}^{\mathrm{B3LYP}}$ base on the same Thomas LSDA approach $E_{\mathrm{XC}}^{\mathrm{LSDA}} = f(\rho)$ and Beckes gradient correction $\Delta E_{\mathrm{XC}}^{\mathrm{BP86}} = f(\rho, |\nabla\rho|)$, but differ in their treatment of correlation energy E_{C}.

$$E_{\mathrm{XC}}^{\mathrm{BP86}} = E_{\mathrm{XC}}^{\mathrm{LSDA}} + a_{\mathrm{X}} \Delta E_{\mathrm{X}}^{\mathrm{B88}} + a_{\mathrm{C}} E_{\mathrm{C}}^{\mathrm{P86}}$$

$$E_{\mathrm{XC}}^{\mathrm{B3LYP}} = E_{\mathrm{XC}}^{\mathrm{LSDA}} + a_0 (E_{\mathrm{X}}^{\mathrm{HF}} - E_{\mathrm{X}}^{\mathrm{LSDA}}) + a_{\mathrm{X}} \Delta E_{\mathrm{X}}^{\mathrm{B88}} + a_{\mathrm{C}} E_{\mathrm{C}}^{\mathrm{LYP}}$$

where, in both, $\Delta E_{\mathrm{X}}^{\mathrm{B88}} = f(\rho, |\nabla\rho|)$ and $\Delta E_{\mathrm{C}}^{\mathrm{LYP}} = f(\rho, |\nabla\rho|)$. In praxis, one major difference between BP86 and B3LYP is that RI-J and MARI-J approximations can be applied to non-hybrid functionals, only.

Computational Methods

In general, all calculations were set up in the same basic way. The ^{125}I incorporating structures to investigate for possibly occuring Coulomb explosions were optimized to ground state geometries and electronic energies. The special isotope mass of ^{125}I was ignored here. Then, ^{125}I was replaced by ^{125}Te, and the new ground state geometries and electronic energies were calculated to obtain the systems to start with. Different charged states of these systems were then optimized, to give either still bound molecular structures, or separate fragments repelling each other. The nature of all fragments obtained was further investigated by applying wave function analysis techniques, especially Mulliken population analysis, and assorting charges to fragments. All fragments were then tested for stability by separate optimization isolated from external fields.

All calculations were carried out within the TURBOMOLE V5-7 [27] program. Generally, DFT was chosen to apply because it gives results—considering molecular geometries and, qualitatively, relative electronic energies—accurate enough for the given problem, while scaling less than quadratically with

the size of the molecular system (i.e. the number of basis functions). Essentially no dynamic methods were applied, because of the expected loss in computational speed and their irrelevancy to the given task.

To approach the more complex bioorganic systems, it first had to be ensured that DFT is a viable level of theory for quantum chemical investigations of Coulomb explosions in general, and, second, whether one could draw any connections between calculated electronic structures and observed fragmentation patterns. Therefore, the Coulomb explosion fragmentation of the simple iodides $CH_3{}^{125}I$, $C_2H_5{}^{125}I$, and n-$C_3H_7{}^{125}I$ were investigated by the methods explained above. To test the results to obtain for consistency, different exchange functional types, BP86 and B3LYP, and different basis sets, i.e. SVP and ^{125}Te relativistic effective core potential ecp-46-mwb [28], SVPall, TZVP/^{125}Te ecp-46-mwb, and TZVPall were used for $CH_3{}^{125}Te$. For $C_2H_5{}^{125}Te$ and n-$C_3H_7{}^{125}Te$ only SVP/^{125}I ecp-46-mwb [29] and SVP/^{125}I ecp-46-mwb, respectively, were used, because molecular geometries and fragmentation patterns were found to be very similar, if not identical for larger basis sets (see results sections). For all calculations using the BP86 functional, the RI-J option was chosen to give faster performance.

Because ^{125}I is inserted in DNA via ^{125}IUdR, the systems of radiomedical relevance to investigate were chosen to be ^{125}Iodouracil, ^{77}Bromouridine-5′-monophosphate, and the isolated ^{125}Iodouridine-5′-monophosphate−Adenosine-5′-monophosphate nucleotide pair. Further, a 10 base pair sequence of alternating Adenine and Thymine, with one Thymine unit replaced by ^{77}Bromouracil, was also possible to investigate. All calculations on the bioorganic systems were carried out using the RI-J option. Additionally, the fast multipole method option MARI-J was applied to the calculation of DNA.

Several technical problems were encountered when investigating the bioorganic systems. First, all calculations on systems combining the heavy elements ^{125}I and ^{125}Te, and phosphorous acid mono- or diester groups did only converge slowly. Standard geometry optimization techniques did not give any stable iodides, in contrast to numerous experimental experience. This was observed for several non-hybride functionals, but not for HF calculations. Although the reasons remain unknown, DFT using non-hybride functionals appears to be not applicable to systems of that kind. To overcome this, ^{125}I and ^{125}Te were replaced by their lighter homologous ^{77}Br and ^{77}Se, respectively, since ^{77}Br is an Auger emitter, too, and these systems seem not to be affected by the problem discussed above. Furthermore, ^{77}Br and ^{77}Se qualitatively exhibit a very similar chemical behavior, and were therefore used for the investigations on the ^{77}Bromouridine-5′-monophosphate, ^{125}Iodouridine-5′-monophosphate−Adenosine-5′-monophosphate nucleotide pair, and DNA system instead of ^{125}I and ^{125}Te.

Second, it appeared to be impossible to carry out any calculation on the 10 base pairs DNA double strand in its proposed form. This is due to the negative charge located on every phosphorous acid diester group, which would repel the two single strands so strongly that they could not remain connected by hydrogen bonds. Physiologically, these negative charges would be compensated by larger histone protein complexes rich in basic—and therefore positively charged—amino acids as Lysine and Arginine, to which DNA binds forming the nucleosome DNA superstructure. Explicitly treating solvation effects or countercharges seems to be impossible. For placing counterions around the DNA sequence one would need a reasonable geometry to start with, which does not exist, and geometry optimization would not succeed because of the general flexibility of the investigated system. Conductor-like screening models like COSMO can not be applied simply because of the size of the system, increasing the computational cost by some orders of magnitude, and would not compensate charge in any way. Thus, each phosphorous acid diester group was saturated with one proton to achieve neutrality. This is supposed to have the least impact on computational speed and molecular geometry, and seems to be without any option within the DFT method applied. To compare results, the same treatment was applied to the ^{125}Iodouridine, and ^{125}Iodouridine-5′-monophosphate−Adenosine-5′-monophosphate nucleotide pair system.

Results

Below are the obtained results of the RI-DFT BP86 and DFT B3LYP calculations on the alkyiodides $CH_3{}^{125}I$, $C_2H_5{}^{125}I$, and n-$C_3H_7{}^{125}I$. The pictures herein give the results of the RI-DFT BP86 calculations.

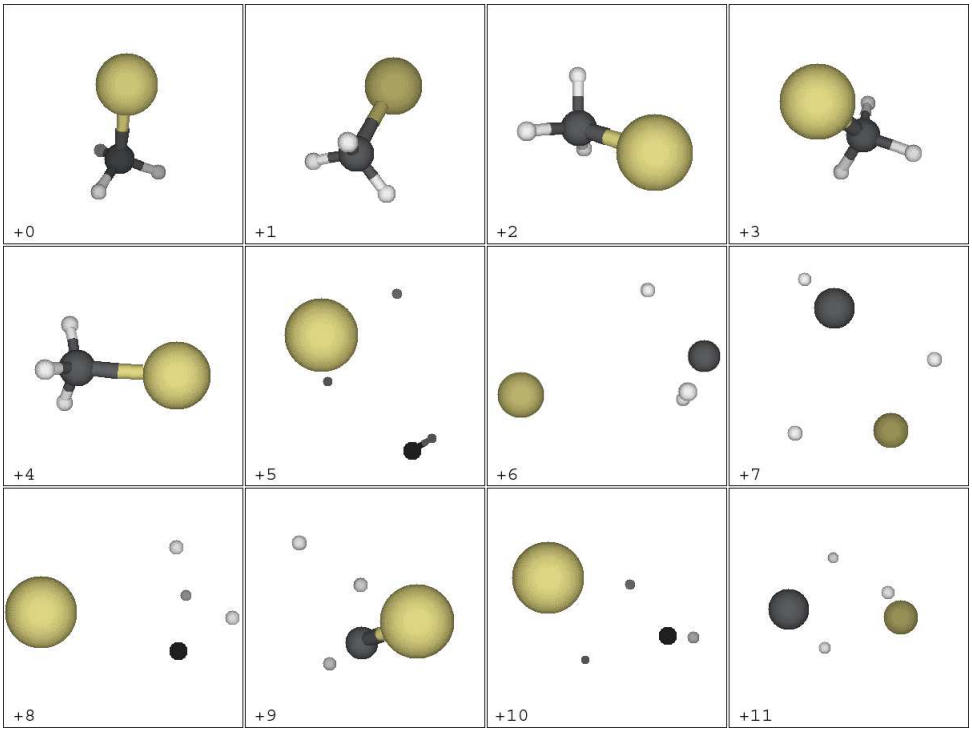


Figure 1: Picture sequence of $CH_3{}^{125}Te$ as a function of charge

Table 1: $CH_3{}^{125}I$ fragments as a function of total charge Q,
RI-DFT BP86, for different basis sets

Q	SVP/ecp-46-mwb	SVPall	TZVP/ecp-46-mwb	TZVPall
+1	$^3CH_3Te^+$	$^3CH_3Te^+$	$^3CH_3Te^+$	$^3CH_3Te^+$
+2	$^2CH_3Te^{2+}$	$^2CH_3Te^{2+}$	$^2CH_3Te^{2+}$	$^2CH_3Te^{2+}$
+3	$^1CH_3Te^{3+}$	$^1CH_3Te^{3+}$	$^1CH_3Te^{3+}$	$^1CH_3Te^{3+}$
+4	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2CH_2^+ + {}^3Te^{2+} + H^+$
+5	$^3CH^+ + {}^1Te^{2+} + 2H^+$	$^3CH^+ + {}^1Te^{2+} + 2H^+$	$^3CH_2^{2+} + {}^1Te^{2+} + H^+$	$^3CH_2^{2+} + {}^1Te^{2+} + H^+$
+6	$^2C^+ + {}^1Te^{2+} + 3H^+$	$^2C^+ + {}^1Te^{2+} + 3H^+$	$^2C^+ + {}^1Te^{2+} + 3H^+$	$^2C^+ + {}^1Te^{2+} + 3H^+$
+7	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^2C^+ + {}^2Te^{3+} + 3H^+$
+8	$^1C^{2+} + {}^2Te^{3+} + 3H^+$	$^1C^{2+} + {}^2Te^{3+} + 3H^+$	$^1C^{2+} + {}^2Te^{3+} + 3H^+$	$^1C^{2+} + {}^2Te^{3+} + 3H^+$
+9	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C^{2+} + {}^1Te^{4+} + 3H^+$
+10	$^2C^{3+} + {}^1Te^{4+} + 3H^+$	$^2C^{3+} + {}^1Te^{4+} + 3H^+$	$^2C^{3+} + {}^1Te^{4+} + 3H^+$	$^2C^{3+} + {}^1Te^{4+} + 3H^+$
+11	$^2C^{3+} + {}^2Te^{5+} + 3H^+$	$^2C^{3+} + {}^2Te^{5+} + 3H^+$	$^2C^{3+} + {}^2Te^{5+} + 3H^+$	$^2C^{3+} + {}^2Te^{5+} + 3H^+$

Table 2: $CH_3{}^{125}I$ fragments as a function of total charge Q,
DFT B3LYP, for different basis sets

Q	SVP/ecp-46-mwb	SVPall	TZVP/ecp-46-mwb	TZVPall
+1	$^3CH_3Te^+$	$^3CH_3Te^+$	$^3CH_3Te^+$	$^3CH_3Te^+$
+2	$^2CH_3Te^{2+}$	$^2CH_3Te^{2+}$	$^2CH_3Te^{2+}$	$^2CH_3Te^{2+}$
+3	$^1CH_3Te^{3+}$	$^1CH_3Te^{3+}$	$^1CH_3Te^{3+}$	$^1CH_3Te^{3+}$
+4	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2CH_2^+ + {}^3Te^{2+} + H^+$
+5	$^1CH^+ + {}^3Te^{2+} + 2H^+$	$^1CH^+ + {}^1Te^{2+} + 2H^+$	$^3CH^+ + {}^3Te^{2+} + 2H^+$	$^1CH^+ + {}^3Te^{2+} + 2H^+$
+6	$^2C^+ + {}^3Te^{2+} + 3H^+$	$^1CH^+ + {}^2Te^{3+} + 2H^+$	$^3C + {}^2Te^{3+} + 3H^+$	$^2C^+ + {}^1Te^{2+} + 3H^+$
+7	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^3CH^+ + {}^1Te^{4+} + 2H^+$
+8	$^1C^{2+} + {}^2Te^{3+} + 3H^+$	$^1C^{2+} + {}^2Te^{3+} + 3H^+$	$^2C^+ + {}^1Te^{4+} + 3H^+$	$^2C^+ + {}^1Te^{4+} + 3H^+$
+9	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C^{2+} + {}^1Te^{4+} + 3H^+$
+10	$^2C^{3+} + {}^1Te^{4+} + 3H^+$	$^1C^{2+} + {}^2Te^{5+} + 3H^+$	$^1C^{2+} + {}^2Te^{5+} + 3H^+$	$^1C^{2+} + {}^2Te^{5+} + 3H^+$
+11	$^2C^{3+} + {}^2Te^{5+} + 3H^+$	$^2C^{3+} + {}^2Te^{5+} + 3H^+$	$^2C^{3+} + {}^2Te^{5+} + 3H^+$	$^2C^{3+} + {}^2Te^{5+} + 3H^+$

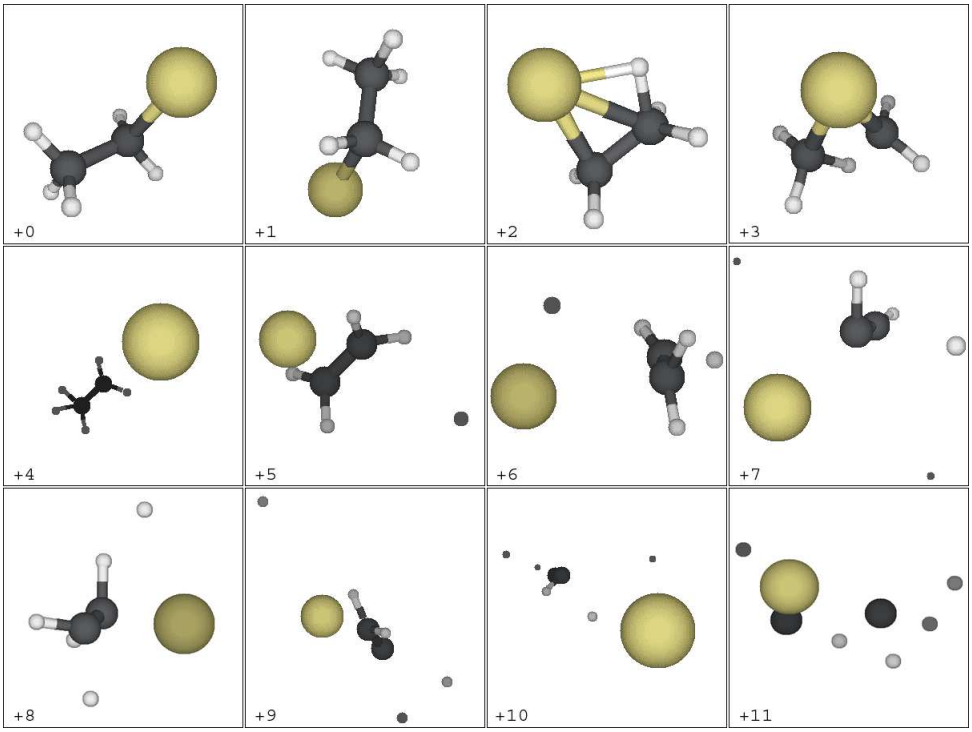


Figure 2: Picture sequence of $C_2H_5{}^{125}Te$ as a function of charge

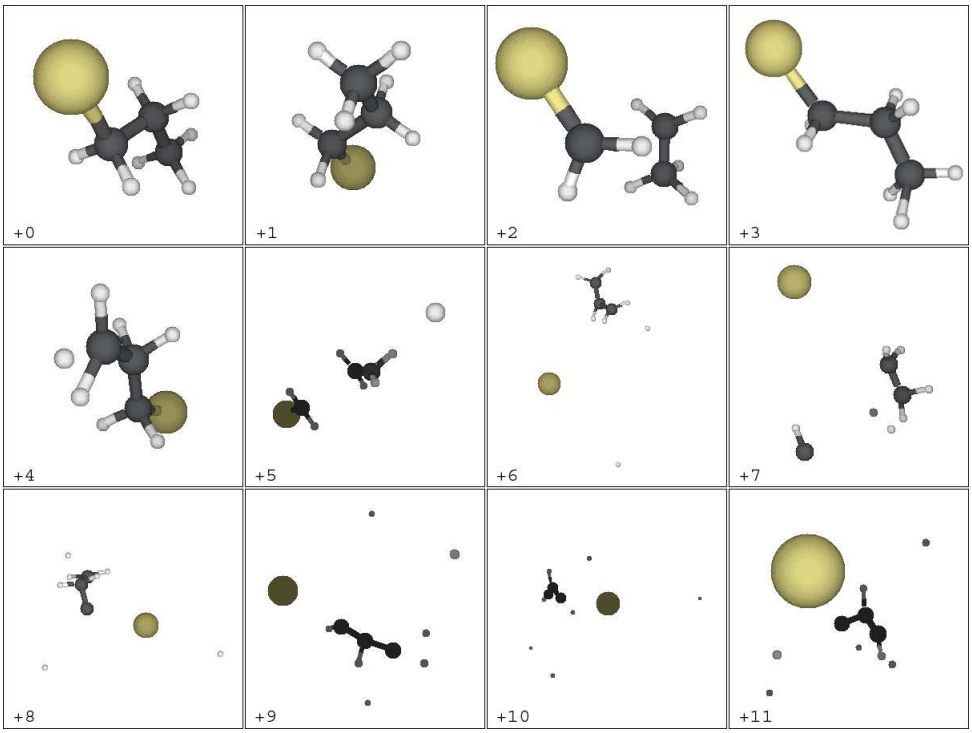


Figure 3: Picture sequence of $C_3H_7{}^{125}Te$ as a function of charge

Table 3: ^{125}I-alkyliodide fragments as a function of total charge Q,
RI-DFT BP86 SVP/ecp-46-mwb

Q	$CH_3{}^{125}Te$	$C_2H_5{}^{125}Te$	n-$C_3H_7{}^{125}Te$
+1	$^3CH_3Te^+$	$^3C_2H_5Te^+$	$^3C_3H_7Te^+$
+2	$^2CH_3Te^{2+}$	$^2C_2H_4TeH^{2+}$	$^1C_2H_5^{2+} + {}^2CH_2Te^+$
+3	$^1CH_3Te^{3+}$	$^1CH_2TeCH_3^{3+}$	$^1C_3H_7Te^{3+}$
+4	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2C_2H_5{}^{2+} + {}^3Te^{2+}$	$^2C_3H_7Te^{4+}$
+5	$^3CH^+ + {}^1Te^{2+} + 2H^+$	$^1C_2H_4^{2+} + {}^1Te^{2+} + H^+$	$^1C_2H_4^{2+} + {}^1CH_2Te^{2+} + H^+$
+6	$^2C^+ + {}^1Te^{2+} + 3H^+$	$^2C_2H_3^{2+} + {}^3Te^{2+} + 2H^+$	$^3C_3H_5^{2+} + {}^1Te^{2+} + 2H^+$
+7	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^3C_2H_2^{2+} + {}^3Te^{2+} + 3H^+$	$^1CH^+ + {}^1C_2H_4^{2+} + {}^1Te^{2+} + 2H^+$
+8	$^1C^{2+} + {}^2Te^{3+} + 3H^+$	$^1C_2H_3^{3+} + {}^2Te^{3+} + 2H^+$	$^1C_3H_4^{2+} + {}^2Te^{3+} + 3H^+$
+9	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C_2H_2^{2+} + {}^1Te^{4+} + 3H^+$	$^1C_3H_3^{3+} + {}^1Te^{2+} + 4H^+$
+10	$^2C^{3+} + {}^1Te^{4+} + 3H^+$	$^2C_2H^{2+} + {}^1Te^{4+} + 4H^+$	$^1C_3H_2^{2+} + {}^2Te^{3+} + 5H^+$
+11	$^2C^{3+} + {}^2Te^{5+} + 3H^+$	$2{}^2C^+ + {}^1Te^{4+} + 5H^+$	$^1C_3H_2^{2+} + {}^1Te^{4+} + 5H^+$

Table 4: ^{125}I-alkyliodide fragments, as a function of total charge Q
DFT B3LYP SVP/ecp-46-mwb

Q	$CH_3{}^{125}Te$	$C_2H_5{}^{125}Te$	n-$C_3H_7{}^{125}Te$
+1	$^3CH_3Te^+$	$^3C_2H_5Te^+$	$^3C_3H_7Te^+$
+2	$^2CH_3Te^{2+}$	$^1C_2H_5^+ + {}^2Te^+$	$^1C_2H_5^+ + {}^2CH_2Te^+$
+3	$^1CH_3Te^{3+}$	$^2C_2H_5^{2+} + {}^2Te^+$	$^1C_3H_7Te^{3+}$
+4	$^2CH_2^+ + {}^3Te^{2+} + H^+$	$^2C_2H_5^{2+} + {}^3Te^{2+}$	$^2C_3H_7Te^{4+}$
+5	$^1CH^+ + {}^3Te^{2+} + 2H^+$	$^1C_2H_4^{2+} + {}^1Te^{2+} + H^+$	$^1C_2H_4^{2+} + {}^3CH_2Te^{2+} + H^+$
+6	$^2C^+ + {}^3Te^{2+} + 3H^+$	$^2C_2H_3^{2+} + {}^3Te^{2+} + 2H^+$	$^1CH_2^{2+} + {}^2C_2H_4^{2+} + {}^1Te^{2+} + H^+$
+7	$^2C^+ + {}^2Te^{3+} + 3H^+$	$^1C_2H_2^{2+} + {}^3Te^{2+} + 3H^+$	$^1CH^+ + {}^1C_2H_4^{2+} + {}^3Te^{2+} + 2H^+$
+8	$^1C^{2+} + {}^2Te^{3+} + 3H^+$	$^1C_2H_3^{3+} + {}^2Te^{3+} + 2H^+$	$^1C_3H_4^{2+} + {}^2Te^{3+} + 3H^+$
+9	$^1C^{2+} + {}^1Te^{4+} + 3H^+$	$^1C_2H_2^{3+} + {}^2Te^{3+} + 3H^+$	$^1C_3H_3^{2+} + {}^2Te^{3+} + 4H^+$
+10	$^2C^{3+} + {}^1Te^{4+} + 3H^+$	$^2C_2H^{2+} + {}^1Te^{4+} + 4H^+$	$^1C_3H_3^{3+} + {}^2Te^{3+} + 4H^+$
+11	$^2C^{3+} + {}^2Te^{5+} + H^+$	$2{}^2C^+ + {}^1Te^{4+} + 5H^+$	$^1C_3H_2^{2+} + {}^1Te^{4+} + 5H^+$

Table 5: ^{125}Te−C bond lenghts as a function of charge Q for stable alkyliodides, in Å,
RI-DFT BP86 SVP/ecp-46-mwb

Q	$CH_3{}^{125}Te$	$C_2H_5{}^{125}Te$	$C_3H_7{}^{125}Te$
0	2.17	2.20	2.19
+1	2.13	2.17	2.17
+2	2.04	2.16/2.44	–
+3	1.98	2.16	2.13
+4	–	–	2.19

Table 6: ^{125}Te−C bond lenghts as a function of charge Q for stable ^{125}TeUdR systems, in Å,
RI-DFT BP86 SVP/ecp-46-mwb

Q	^{125}TeUdR
0	2.08
+1	2.04
+2	2.03
+3	2.05
+4	2.14
+5	2.34

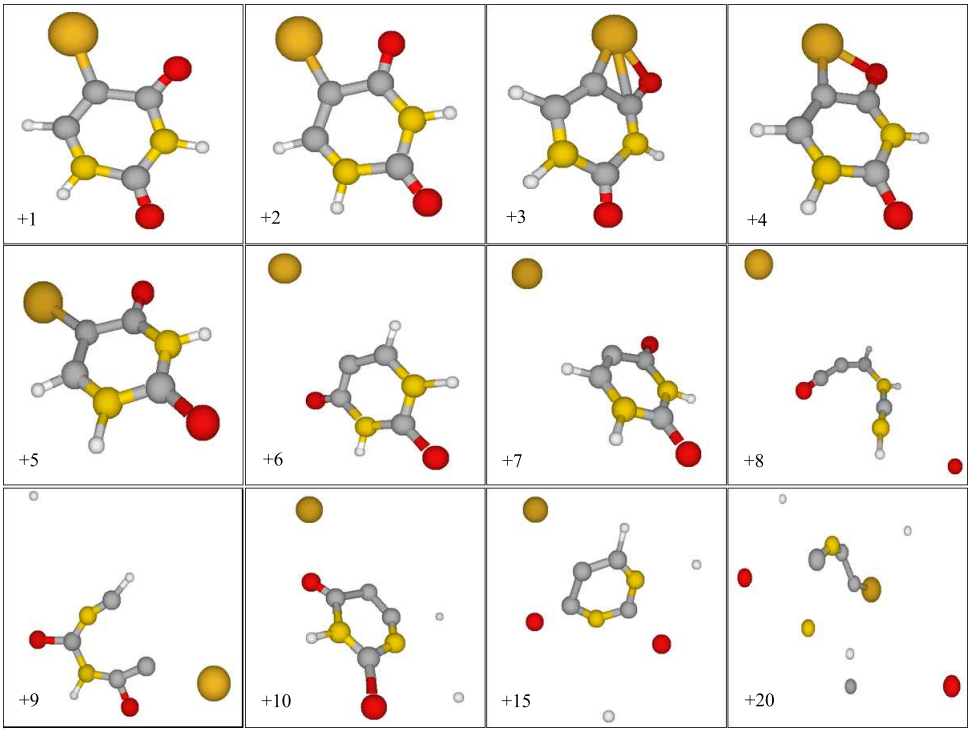


Figure 4: Picture sequence of ^{125}TeUdR as a function of charge

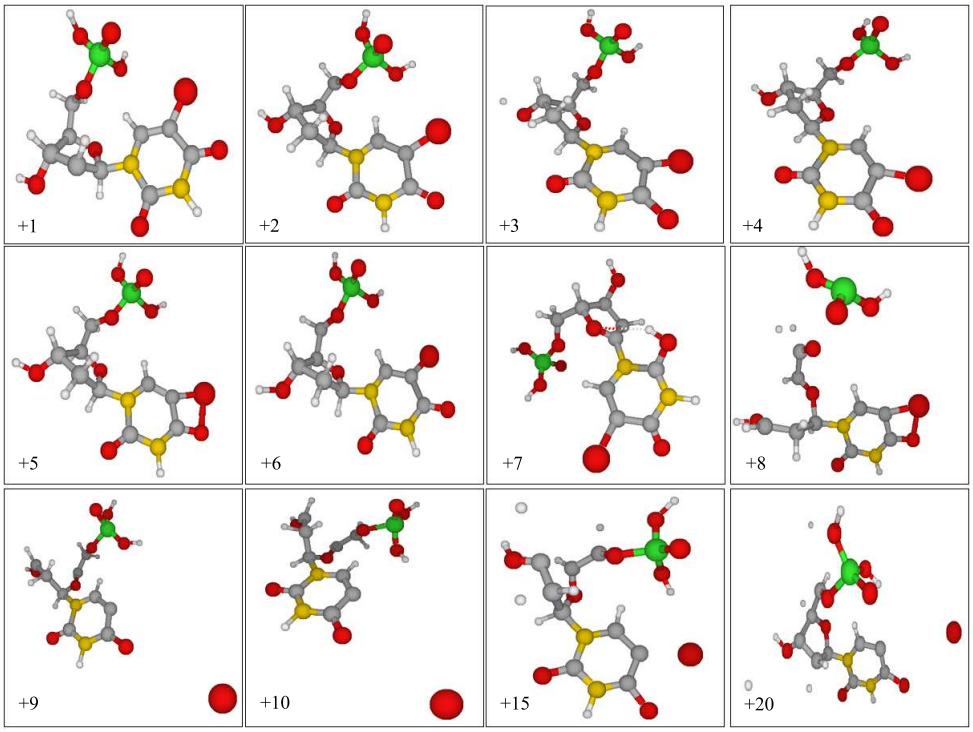


Figure 5: Picture sequence of Thymine nucleotides as a function of charge

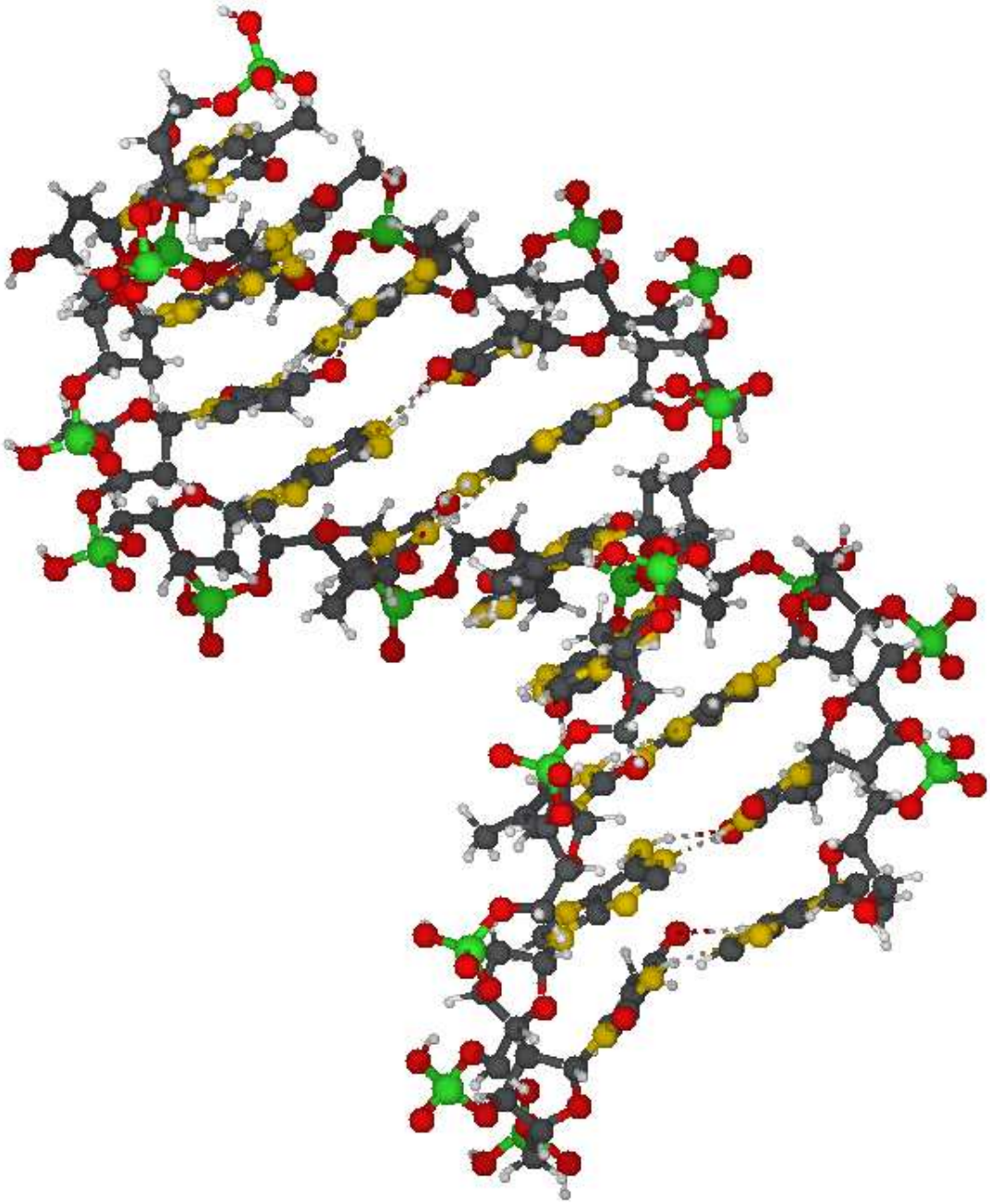


Figure 6: Picture of a DNA molecule

Discussion

Alkyliodides

The results obtained from the mass spectrometry investigations of Coulomb explosion fragmentation of $CH_3{}^{125}I$ and $C_2H_5{}^{125}I$ [2, 18] have been verified in a way that qualitatively most experimentally yielded fragments were found in the current work. However, comparison of experimental data with calculations must be handled with care, because of observed secondary field ionization processes inside MS devices [19]. The obtained results can not answer the question about the relative abundances of the fragments because the method of the calculations carried out here can not simulate the Auger process itself.

Table 1 give the obtained fragmentations of $CH_3{}^{125}I$, using the exchange functional BP86. The fragmentation patterns are identical for all charges and basis sets, i.e. SVP/ecp-46-mwb, i.e. SVPall, TZVP/ecp-46-mwb, and TZVPall. The only exception was found for $CH_3{}^{125}Te^{5+}$ for TZVP/ecp-46-mwb, where one additional proton appears to be bound to the carbohydrate fragment.

Table 2 give the obtained fragmentations of $CH_3{}^{125}I$, using the exchange functional B3LYP. All basis sets give identical fragmentation patterns for systems up to a total charge of +5, except of the different multiplicities for the $CH_3{}^{125}I^{5+}$ systems. For higher charged systems different basis sets yield different fragmentation patterns. Generally, basis sets bigger than SVP/ecp-46-mwb result in smaller charges on the carbon atom, as observed for $CH_3{}^{125}I^{6+}$ with SVPall and TZVP/ecp-46-mwb, and $CH_3{}^{125}I^{7+}$ for TZVPall. With increasing sizes of the basis sets applied, that is SVP/ecp-46-mwb $<$ SVPall $<$ TZVP/ecp-46-mwb $<$ TZVPall, ^{125}Te appears to bear higher charges. Whereas all basis sets give a $^{125}Te^{5+}$ cation for the maximum total charge calculated, higher charges appear for lower total charges with increasing basis set size.

For both exchange functionals the effect of applied ECPs—which divide all electrons into two groups, one valence shell and one core shell for all others—onto the observed fragmentation patterns can not be ascertained. One could expect that the ^{125}Te-ECP should result in lower charges on ^{125}Te. In fact it appears that using ECPs or not does not have any impact on the charge distributions on ^{125}Te cations, whereas this seems to be affected only by basis set size. However, for BP86 no differences between ECP and all-electron basis sets were observed.

Tables 3 and 4 give the obtained fragmentation patterns of the alkyliodides $CH_3{}^{125}I$, $C_2H_5{}^{125}I$, and n-$C_3H_7{}^{125}I$, using the different exchange functionals BP86 and B3LYP. The investigated $C_2H_5{}^{125}I$ structures, using the BP86 functional, converged up to a charge of +3. At higher charges the further fragmentation started first by cutting off ^{125}Te. Successive charging the $C_2H_5{}^{125}Te$ system undergoes loss of a growing number of protons until complete fragmentation.

B3LYP calculations show much earlier fragmentation at a charge of +2 and +3 for $C_2H_5{}^{125}I$. The following higher charged systems give similar fragmentation patterns for both functionals, especially in $C_2H_5{}^{125}Te$ for ethyl ions and protons. The only difference between these two functionals is in the charges of their fragments.

In case of the n-$C_3H_7{}^{125}Te$ both functionals (B3LYP, BP86) give almost the same results. As in calculations on $CH_3{}^{125}Te$ the same fragments occur in different charges, and the fragmentation patterns are even almost equal.

The BP86 functional seems to exhibit a more systematic fragmentation pattern (see below), concerning different basis sets, than B3LYP, because of the inconsistent appearing of fragments like CH ions and several ions of ^{125}Te and Carbon for different basis sets (tables 1 and 2). These ions differ in charges and multiplicities for systems of equal total charge, but calculated with different functionals. All systems undergo fragmentation by similar principles, like loss of ^{125}Te cations and protons with the very beginning

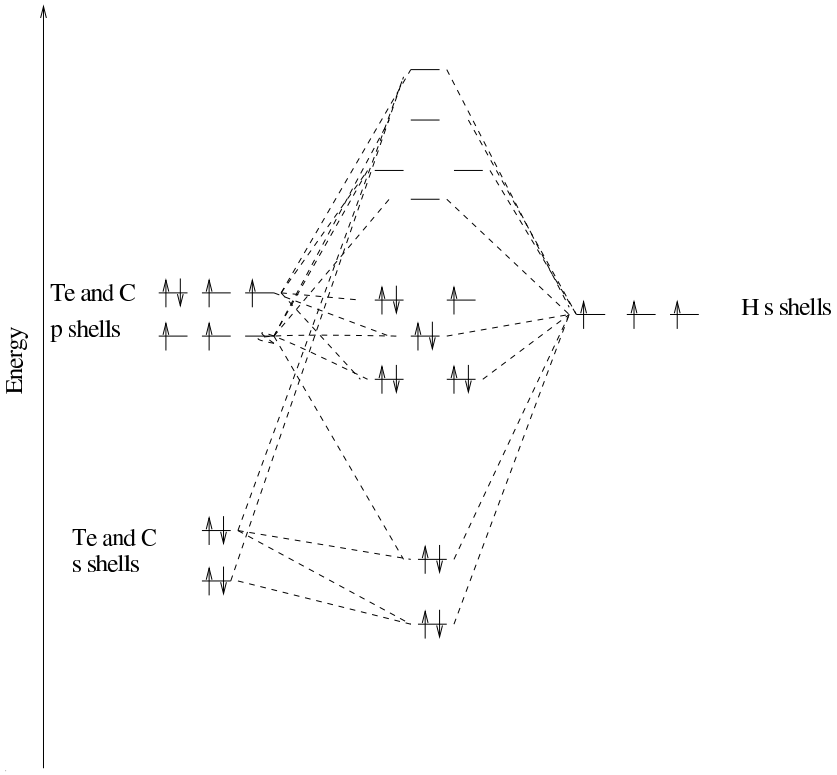


Figure 7: Qualitative MO scheme of $CH_3{}^{125}Te$

of fragmentation. These atomic ions can carry relatively high charges out of the system, whereas they provide quite low bonding energies to the formerly bound molecule. In all cases the carbon backbone remains intact, though charged and successively loosing more protons, for almost all calculated total charges. Few exceptions are observed e.g. for n-$C_3H_7{}^{125}Te^{2+}$, n-$C_3H_7{}^{125}Te^{5+}$, n-$C_3H_7{}^{125}Te^{6+}$, and n-$C_3H_7{}^{125}Te^{7+}$ for both BP86 and B3LYP. However, all fragments turned out to be stable when calculated separately.

To interpret obtained results it is an appropriate way to use MO schemes to derive fragmentation patterns for smaller molecules. The $CH_3{}^{125}Te$ MO scheme gives an easy understanding of its fragmentation and in basic qualitative aspects this could be valid to higher alkyliodides, too. In table 3 the $^{125}Te-C$ bond length became shorter by charging $CH_3{}^{125}Te$. In classical way one would assume the other direction because of more repulsive interactions in cations—an increasing bond length. This was not observed.

In Fig.7 is a representing MO scheme of $CH_3{}^{125}Te$ in C_{3v} symmetry and full occupation of binding MOs by its 13 electrons from their valence shells. If one up to three electrons are removed this scheme develops right behavior of the $^{125}Te-C$ bond length—it becomes shorter (see table 5). It could be stated that the repulsion between the methyl group and the non-binding electrons from ^{125}Te reduces when electrons are removed out of this area. Attractive interaction towards the binding electrons from the nuclei of ^{125}Te and Carbon is another possible reason for a shortened bond length. For a normal $^{125}Te-C$ single bond length one has to derive values from covalent radii of both atoms [31].

When four electrons are removed the repulsive energy of the nuclei increases due to the loss of binding energy, and the molecule fragments just as seen before on the pictures. From a charge of +4 on, $CH_3{}^{125}Te$ is not stable.

Going on removing electrons one recognizes that an interesting constellation appears at a charge of

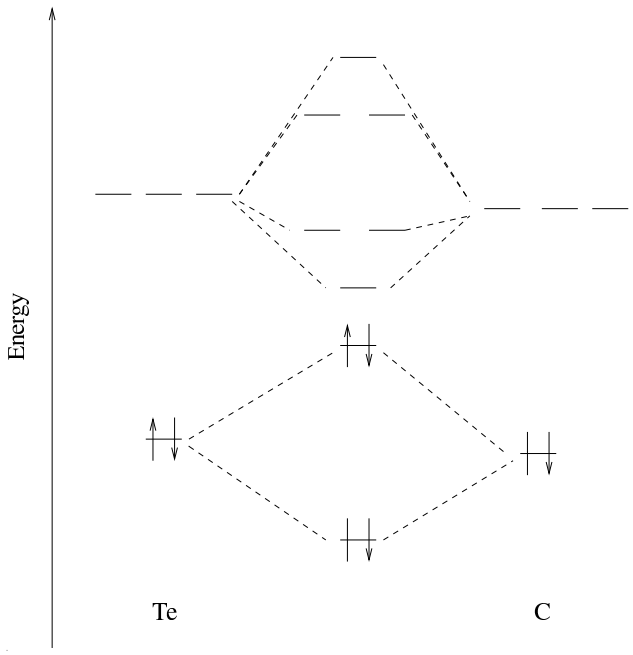


Figure 8: Qualitative MO scheme of $C^{125}Te^{+6}$

+9. At this point all hydrogens have left and all p shells from ^{125}Te and Carbon are empty, and only the s shells are occupied. In Fig.8 this $C^{125}Te^{+6}$ ion is seen. This is an artifact because of its σ MO levels, which have fully occupied antibinding and binding MOs. The antibinding energy is higher than the binding. Thus $C^{125}Te^{+6}$ can not exist, it has to be an artifact. This could also be possible because of electrostatic reasons. For higher charged systems one can develop an interpretation for the $CH_3{}^{125}$Te fragmentation pattern, calculated with BP86. As presented in table 1 one sees, that in every calculation on this molecule, using different basis sets, the completely fragmented systems have got the same charge distribution pattern. To understand this, obviously, the MOs have to be replaced by the AOs of ^{125}Te and Carbon. If considered that the ^{125}Te and Carbon p shells are on a similar energetic level (see Fig.8), one recognizes that a mechanism of alternating ionization can be established. First two ^{125}Te electrons are removed to get to an equal electronic configuration as Carbon has. Then this mechanism starts by removing an electron from the Carbon (total charge +6), then from ^{125}Te, then from Carbon again, and so on, until the maximum calculated total charge of +11 is reached.

For higher alkylioides the obtained fragmentation patterns could be equal because of their similar MO levels. Of course many other alkyl group fragments should appear at bigger systems. But in general the same ions like ^{125}Te and protons would be first obtained in similar calculations.

All $CH_3{}^{125}$Te and $C_2H_5{}^{125}$Te fragments obtained in this work were found experimentally [2, 18] by mass spectrometry investigations (i.e. CH_3Te^+, CH_3Te^{2+}, CH_2Te^+, CH^+, CH_2^+, $C_2H_5^+$, $C_2H_5^{2+}$, $C_2H_4^+$, $C_2H_4^{2+}$, $C_2H_3^+$, $C_2H_3^{2+}$, $C_2H_2^{2+}$, C_2H^+, C_2H^{2+}, and atomic ions). More fragments were found in experiments than in this work, because of secondary ionization [19] and ion recombination reactions in the mass spectrometer.

Concluding, it can be stated that DFT is a viable technique for calculating extremely charged systems by means of quantum chemistry. Basis sets as small as SVP, using effective core potentials for heavy elements like Te, appear to give satisfying results compared to much larger and though more time consuming basis sets like TZVPall. Thus small basis sets for DFT calculations can be sufficient to derive molecular ground state geometries.

Even conceptually different exchange functionals as BP86 and B3LYP seem not to have any impact on the fragmentation patterns due to Coulomb explosions. For very simple systems one can understand fragmentation of highly charged molecular systems from their MO energies and occupations.

Bioorganic Compounds

^{125}Iodouracil (^{125}IUdR) and ^{77}Bromouridine-5$'$-monophosphate were chosen to be the systems to investigate because ^{125}I is commonly inserted into mammalian DNA by ^{125}IUdR. The systems were calculated with charges of +1 to +10, +15, and +20. Similar calculations were carried out by Pomplun [21, 30] on ^{125}IUdR with the semiempirical method PM3.

^{125}Iodouracil

Table 6 and Fig.4 give the results of the calculations on ^{125}TeUdR. In good agreement to Pomplun's [21, 30] results, stable molecules were found for lower charges. Even the same deformations of the Tellurium−Carbon bond were found. Increasing charge turns ^{125}Te towards the neighbored O of the keto function. Up to charge of +5 Pomplun found stable molecules in relative normal conformations. For +6 charge enormous deformations of the aromatic ring were observed, but this is probably due to the semiempirical PM3 level of theory applied, which is not able to describe bond dissociation properly. Instead of deforming at higher charges, ^{125}TeUdR fragments into two parts in this work. From +6 to very high charges one finds, like in case of the alkyliodides, similar fragmentation patterns: First there will be ^{125}Te and Hydrogen ions emitted, and then the aromatic ring will be destroyed. A possible reason for the stability of the ring system is, of course, its aromaticity, contributing additional binding energy to this ring system. At high charges ^{125}TeUdR fragments in several ions. In cases of very high charges artifacts could possibly be obtained, as e.g. the opened ring group binding ^{125}Te (charge +20), or retained ring groups (charges +10, +15), though ring fragmentation already occured at charges of +8 and higher.

Similar to the former discussion about the alkyliodides there are ^{125}Te and Hydrogen as first fugitive components. The size of this molecule makes it hard to build a MO scheme, so it could be considered that the occupied MOs rearrange in a special way such that artifacts appear. As in Pomplun's work [21], the ^{125}Te−C bond length found here shortens up to charge +2, and then increases with growing charge (table 6). Pomplun found that the ^{125}Te−C distance in +5 charged ^{125}TeUdR is about 20% longer than of the +4 charged system. Looking at table 6, here the bond length increases only about 10% from +4 to +5 charged ^{125}TeUdR. Comparing with Pomplun's results [21], it seems that different methods for calculating Coulomb explosions of bioorganic systems yield comparable results for at least lower charged systems. So it could be stated that DFT method is one appropriate way for theoretical DNA investigation.

Uridine Nucleotide

In several DFT calculations the ^{125}Te incorporating uridine nucleotides could not converge because of the ^{125}Te separating from the thymine nucleotide group. To keep the results comparable a similar element was chosen to replace the not binding ^{125}Te. So ^{77}Se with thymine nucleotide was investigated instead. It gave comparable geometry parameters like similar C−^{77}Se bond length and angles.

The investigation of the uridine nucleotides with ^{77}Se inserted lead to the results shown in Fig.5. For lower charged systems, up to a charge of +6, no fragmentation appears, like in case of the ^{125}TeUdR system. At a charge of +7 first protons and ^{77}Se are cut off from the nucleotide. A reason for this might be the ability of this ions to carry positive charges as discussed in the former chapters. In +7 charged nucleotides the structure also deforms in a way that a proton binds at an O from the uracil to form a new

O−H bond, and also undergoes a hydrogen bond to another O from the ribose ring system. The ^{77}Se is also twisted to its neighbored O like in ^{125}TeUdR. This twisting towards its neighbor is even observed in +8 charged system, until the ^{77}Se separates from the nucleotide group in +9 charged system. In every case of fragmentation the C−^{77}Se bond is deformed or cut. For higher charged systems, as +10, +15, and +20, the fragmentation patterns are similar to the lower charged systems.

An interesting point is the C−C bond breaking in the ribose ring system. This is observed for charges of +8 to +10, and +15. At all structures the same bonds are broken in the ribose ring. The reason could be that the two separated parts can handle their charges because of the neighbored O groups, which can stabilize positive charges by their lone pairs. Here, the O can give one of its lone pairs to bear some positive charge of the alkyl groups. So, the calculated fragments with this broken ribose parts could probably be no artifacts. So far this ribose breaking might be a possible route to cause SSB in the DNA.

The uracil base subsystems appeared to be stable in all performed calculations, probably because of its aromatic ring system. This gives additional binding energy to the ring so that fragmentation of the ring system was not observed. This could also be understood by the argument that the whole nucleotide molecule could carry more positive charges than the smaller thymine base with ^{125}Te.

In one case (charge +8) even H_2PO_3 was formed, but its charge and multiplicity was not investigated. Thus this fragment could be a kind of phosphoric acid derivative.

DNA

The 10 base pair DNA double helix was optimized using MARI-*J*-DFT BP86 methods, with a SVP basis set for all atoms (see Fig.6). This system was saturated with one proton for each bridging phosphorous acid diester for electroneutrality reasons (see methods).

RHF-SCF convergence appeared to be relatively fast within an energy treshold of 10^{-6} a.u. (about 100 steps). However, no minimum geometry could be found using the standard geometry optimization techniques of TURBOMOLE. This is, first, possibly due to the overall flexibility of the molecular structure, the two double strands being connected only by hydrogen bonds. Thus the potential energy hypersurface is assumed to exhibit several local minima near the global one. Second, the initial EHT start orbitals were found to be energetically very close to each other, and a manifold of almost degenerate states near the HOMO−LUMO gap must be expected. Therefore, several optimization cycles were carried out until the total energy gradient dropped below 10^{-1} a.u., and the geometry obtained by this was used as a start geometry for the further investigations.

The situation became worse for substituting the methyl group of one thymine base by ^{77}Br, to get a Auger emitter incorporating DNA model system. This system was calculated neutral, and with a positive charge of +15. For both cases, it appeared to be impossible by standard techniques to reach UHF-SCF convergence, even to converge to a set of MOs changing less than about 10^{-4} a.u. in its total electronic energy. HOMO−LUMO gaps dropped below 0.005 a.u. in every calculation, and could not be fixed by any combinations of UHF-SCF or occupation number options in TURBOMOLE, e.g. orbital shift—automatic or manual—or SCF damping methods. One possible reason for that might be the unrestricted nature of the calculations carried out on that system. By any means, this was absolutely necessary because fragmentation, or at least any response to the high charge, was expected.

Conclusions

It was shown that Density Functional Theory (DFT), though a non-dynamical approach, is an appropriate method to calculate the Coulomb explosion of organic and bioorganic compounds. Molecular fragments formed by Auger process induced Coulomb explosions could be predicted, agreeing with experimental

results, and characterized in terms of charge, multiplicity, and stability.

The identified fragments show a qualitative but clear trend in fragmentation reactions—following the much faster Auger process—for different investigated systems. H^+ and ^{125}Te cations, and H^+ and ^{77}Se cations, respectively, were the first components to be repelled from the fragmenting molecule in every single calculation. For the calculations on bioorganic systems containing phosphorous acid mono- or diesters, ^{77}Br was chosen for the Auger emitter instead of ^{125}I, because the latter appeared to be impossible to investigate by DFT techniques. No stable iodide structures were found here, in contrast to various experimental results. Similarly, no fragmentation of highly charged 10 base pair DNA sequences could be calculated successfully.

Assuming that the fragmentation found for simple organic systems, i.e. $CH_3{}^{125}I$, $C_2H_5{}^{125}I$, and n-$C_3H_7{}^{125}I$, and larger systems as ^{77}Br incorporating uridine nucleotides, could be a model for bioorganic compounds in living cells, the Coulomb explosion could in fact be a viable pathway to Auger process induced DNA damage. As seen for uridine, C−C bond breaking in the ribose ring may be, for example, a possible fragmentation reaction. On the other hand, aromatic systems appear to hold great stability against fragmentation due to high charges, as shown by comparable results on the ^{125}TeUdR system, calculated with DFT, HF, and semiempirical PM3 methods.

Acknowledgements

We would like to thank our supervisor Dr. Th. Müller for his patient support and the valuable discussions, more on-demand lectures, which made us understand the physics behind chemistry much better than before. Ivo Kabadshow is especially acknowledged for his help on certain parallel TURBOMOLE problems. We would also like to thank Dr. R. Esser, Dr. J. Grotendorst and Prof. Dr. M. Dolg, University of Cologne, for enabling us to participate in the NIC guest student program 2004, and the ZAM staff and all other guest students for their great company.

References

1. ERC Handbook of Chemistry and Physics, 64th. ed., B232−B316, CRC Press, 1983−1984.
2. T. A. Carlsen, R. M. White, J. Chem. Phys., **1963**, 38, No. 12, 2930−2094.
3. K. G. Hofer, Acta Oncologica, **2000**, 39, No. 6, 651−657.
4. K. G. Hofer, W. Prensky, W. L. Hughes, J. Nat. Cancer Inst., **1969**, 43, 763−773.
5. K. G. Hofer, W. L. Hughes, Radiat. Res., **1971**, 47, 94−109.
6. H. H. Ertl, L. E. Feindegen, H. J. Heiniger, Phys. Med. Biol., **1970**, 15, 447−456.
7. L. E. Feindegen, H. H. Ertl, V. P. Bond, in: H. Ebert (Ed.), Proceedings of the Symposium on Biological Aspects on Radiation Quality, Vienna, IAEA, 1971, 419−430.
8. R. C. Warters, K. G. Hofer, C. R. Harris, J. M. Smith, Curr. Top. Rad. Res. Q, **1977**, 12, 389−407.
9. O. A. Sedelnikova, I. G. Panyutin, A. R. Thierry, R. D. Neumann, J. Nucl. Med., **1998**, 39, 1412−1418.
10. R. F. Martin, W. A. Haseltine, Science, **1981**, 213, 896−898.
11. I. G. Panyutin, R. D. Neumann, Nucleic Acid Res., **1994**, 22, 4979−4982.
12. A. Bishayee, D. V. Rao, L. G. Bouchet, W. E. Bolch, R. W. Howell, Radiat. Res., **2000**, 153, ...
13. V. R. Narra, R. S. Harapanhalli, R. W. Howell, K. S. R. Sastry, D. V. Rao, Radiat. Res., **1994**, 137, 394−399.
14. M. A. Walicka, S. J. Adelstein, A. I. Kassis, Radiat. Res., **1998**, 149, 142−146.
15. D. E. Charlton, J. Booz, Radiat. Res., **1981**, 87, 10-23.
16. E. Pomplun, Int. J. Radiat. Biol., **1991**, 3, 625−642.
17. M. Terrisol, Int. J. Radiat. Biol., **1994**, 33, 279−451.
18. T. A. Carlson, R. M. White, J. Chem. Phys., **1966**, 44, No. 12, 4510−4520.
19. L. Poth, Q. Zhong, J. V. Ford, S. M. Hurley, A. W. Castleman Jr., Chem. Phys., **1998**, 239, 309−315.
20. P. Auger, Comp. Rend., **1925**, 180, 65−68.
21. E. Pomplun, Acta Oncologica, **2000**, 39, No. 6, 673−679.
22. G. Sutmann, personal communication.
23. L. H. Thomas, Proc. Camb. Phil. Soc., **1927**, 23, 542−548.
24. E. Fermi, Rend. Accad. Lincei, **1927**, 6, 602−607.
25. P. A. M. Dirac, Proc. Camb. Phil. Soc., **1930**, 376−385.
26. P. Hohenberg, W. Kohn, Phys. Rev. B, **1964**, 136, 864−871.

27. R. Ahlrichs, M. Bär, M. Häser, H. Horn, C. Kölmel, Chem. Phys. Letters **1989**, 162, 156−...
28. A. Bergner, M. Dolg, W. Kuechle, H. Stoll, H. Preuss, MOl. Phys., **1993**, 80, 1431−...
29. P. Schwerdtfeger, M. Dolg, W. H. E. Schwarz, G. A. Bowmaker, P. D. W. Boyd, J. Chem. Phys., **1989**, 91, 1762−...
30. E. Pomplun, G. Sutmann, Int. J. Rad. Biol., in press.
31. A. F. Holleman, Lehrbuch der Anorganischen Chemie, 101. ed., deGruyter, Berlin, New York, 1995, 1838−1841.

NMR-Quantum Computer Simulation on a Parallel Supercomputer

Nikolas Pomplun

Technical University of Berlin
Institute of Physics

E-mail: pomplun@physik.tu-berlin.de

Abstract: The aim of this work was to simulate the time evolution of a quantum computer system. The simulation code was optimized to run on a parallel computer that provides the needed resources to model the time evolution in a reasonable time. Thereby the attention was focused on two major features. First the Suzuki-Trotter algorithm that was used to model the time evolution and second the adjustment to a specific realisation of such a computer: the Nuclear Magnetic Resonance Quantum Computer.

Introduction

A quantum computer can solve certain computationally hard problems much faster than a classical computer under the circumstances that the algorithm makes use of the quantum parallelism. Thus in order to exploit the huge potential of a quantum computer a simulator is indispensable for gaining further knowledge in particular since theory is far ahead of the experiment. The basic task of a quantum computer simulator is to solve the time dependent Schrödinger-equation for all the qubits involved. For a system of n qubits the dimension of our problem and the number of equations to be solved scales with 2^n. If we have for example 25 qubits we are dealing with $2^{25} = 33.554.432$ equations not considering the amount of memory that would be needed to store the state vector or the operators. Furthermore, by taking the step from an ideal (theoretical) quantum computer to a real physical system we have to take into account a lot of unwanted side-effects as explained later in this text that turn the system into a many-body problem. Hence the problem can no longer be solved analytically. It becomes necessary to approximate the system by numerical calculations. Both of the aspects that are mentioned here demonstrate that it is necessary to use a parallel (still classical) supercomputer. Nowadays simulations with up to 30 qubits can be handled. To make a quantum computer competitive to a contemporary classical computer we would need at least tens to hundred qubits. This also shows the limits of a classical simulation because to store only a state vector of a 250 qubit system it would require as many bytes as there are particles in the universe ($\sim 10^{80}$).

In the theory section some basics of quantum computing will be introduced covering the main aspects of the ideal quantum computer and the time evolution. The differences that have to be considered regarding a real NMR-quantum computer are followed by a closer look at the Suzuki-Trotter algorithm. After a short overview of the structure of the program itself the programming section addresses the issue of how to implement OpenMP and MPI and explains the functioning of the core subroutine that does the actual computation. At last there will be a conclusion showing some results, difficulties and further ideas.

Theory

The Ideal Quantum Computer

Where a classical computer uses bits a quantum computer uses qubits to store its data. A qubit can be thought of as a 2-level system e.g. a spin $\frac{1}{2}$ system that is described by a state vector in a 2-dimensional complex Hilbert space. If we choose the Eigenstates of the z-Pauli-operator $|0\rangle = \binom{1}{0}$ and $|1\rangle = \binom{0}{1}$ as basis states the state of the qubit can be written as

$$|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{1}$$

$$|\alpha|^2 + |\beta|^2 = 1 \quad \alpha, \beta \in \mathbb{C} \tag{2}$$

The parameter α can be assumed real since we can always extract an overall phase that does not contribute to the expectation value. While in classical computers either α or β is exactly equal to 1 in quantum mechanics a superposition of $|0\rangle$ and $|1\rangle$ is possible as far as the normalisation condition (equation 2) holds. Superposition is a special quantum property that e.g. allows faster computation in comparison to a classical computer if we can implement this quantum parallelism in the algorithm. Qubits also have other non-classical properties like entanglement for example that opens up new ways of computation quite different from the well known classical ones.

Since α can be assumed real we can picture the state of a qubit in the so called Bloch-sphere where a state is represented by a point inside the unit sphere determined by 3 real parameters θ, ϕ and r. The states for which $r = 1$ holds are called pure states and are located on the surface of the Bloch-sphere. States inside the Bloch-sphere correspond to mixed states which we will not consider here for reasons of simplicity. Since every additional qubit doubles the number of basis states our system has $\dim = 2^{[\text{no.ofQubits}]}$. For

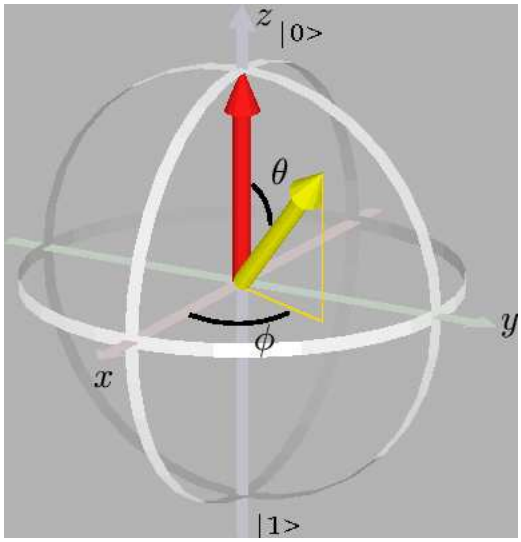


Figure 1: Bloch sphere

a 2-qubit system as it is considered in this work we are dealing with 4 basis states. The state vector is then given by

$$|\Psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \tag{3}$$

$$|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1 \tag{4}$$

Single qubit operations can be pictured as rotations of the Bloch-vector around a certain axis.

In analogy to classical computing, it turns out that there is a universal gate set for a quantum computer that is sufficient to perform every possible operation. One possible set contains the rotations around the x-,y- and $z-$ axis together with the CNOT operation that acts on 2 qubits and flips the spin of the second qubit if the the spin of the first qubit is up (figure 2).

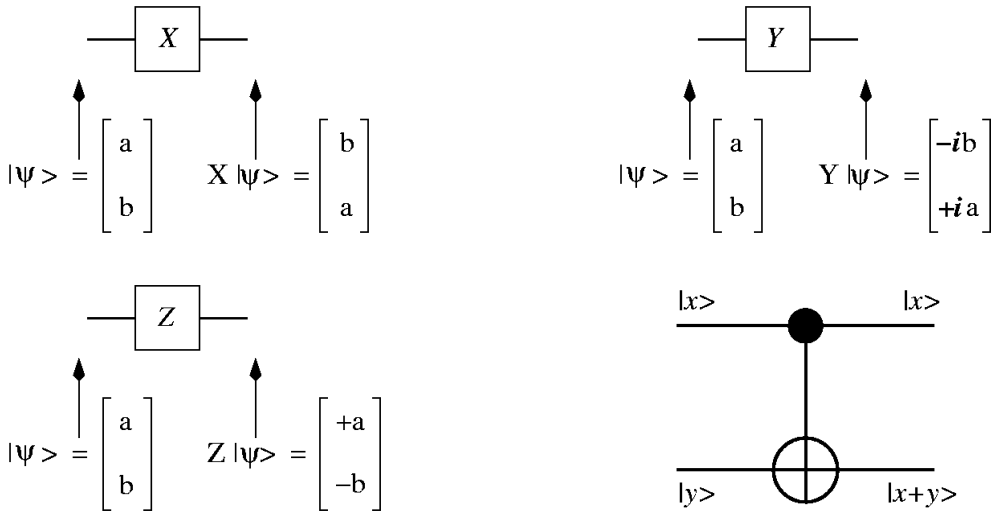


Figure 2: a universal gate set

Time Evolution

The time evolution of a quantum system is governed by the Schrödinger-equation

$$i\frac{d}{dt}|\Psi(t)\rangle = \mathcal{H}(t)|\Psi(t)\rangle \tag{5}$$

Integration yields

$$|\Psi(t+\tau)\rangle = \exp\left(-i\int_t^{t+\tau} d\tau \mathcal{H}(\tau)\right)|\Psi(t)\rangle = U(\tau)|\Psi(t)\rangle = e^{-i\tau\mathcal{H}}|\Psi(t)\rangle \tag{6}$$

where $U(t,t+\tau)$ is a unitary operator the so called time propagator that transforms $|\Psi(t)\rangle$ into $|\Psi(t+\tau)\rangle$. Note that $e^{-i\tau\mathcal{H}(t)}$ is unitary by construction. The most general form of a Hamiltonian describing a quantum spin system reads

$$\mathcal{H}(t) = -\sum_{i,j=1}^{L}\sum_{\alpha=x,y,z} J_{i,j}^{\alpha}(t)S_i^{\alpha}S_j^{\alpha} - \sum_{i=1}^{L}\sum_{\alpha=x,y,z} h_i^{\alpha}(t)S_i^{\alpha} \tag{7}$$

The first factor describes the qubit-qubit interaction between the qubit-pairs while the second factor describes the external influences acting on single qubits. In the case of 2 qubits this expression reduces to

$$\mathcal{H}(t) = -J_{1,2}^{\alpha}(t)S_1^{\alpha}S_2^{\alpha} - \sum_{i=1,2}\sum_{\alpha=x,y,z} h_i^{\alpha}(t)S_i^{\alpha} \tag{8}$$

The Nuclear Magnetic Resonance Quantum Computer

While we can compute the propagation of an ideal quantum computer analytically in the non-ideal case there are additional impacts on our system dependent on the physical realisation of the quantum computer that make the Schrödinger-equation no longer solvable analytically. In addition to that the order of the applied operations matters so that a different order results in different outcomes. In the Nuclear-Magnetic-Resonance computer a whole ensemble of spins enclosed in a suited liquid is used to represent the qubits. There is a static magnetic field along the $z-$axis which splits up the spin-energy-states due to the Zeeman-effect. Each spin rotates with its Larmor frequency around the $z-$axis while it also couples in $z-$direction to the magnetic field of the other spins. So according to the Ising model our Hamiltonian for a freely evolving system reads

$$\mathcal{H}(t) = -J^z_{1,2}(t)S^z_1S^z_2 - h^z_1S^z_1 - h^z_2S^z_2 \tag{9}$$

Single-qubit operations are carried out by short sinusoidal radio-pulses tuned to the Larmor-frequency ω of the qubit. Thus if a radio-pulse $\mathcal{H}_{x,1}(t) = -(h^x_1S^x_1 + h^x_2S^x_2)sin(\omega t)$ is applied to qubit 1 along the $x-$axis the Hamiltonian becomes

$$\mathcal{H}(t) = -J^z_{1,2}(t)S^z_1S^z_2 - h^z_1S^z_1 - h^z_2S^z_2 - (h^x_1S^x_1 + h^x_2S^x_2)\sin(\omega t) \tag{10}$$

$J^z_{1,2}$ and h^α_i are time dependent and can be controlled during the experiment. The duration of the radio-pulses determines the angle about which the spin rotates.

The Suzuki-Trotter-Algorithm

In order to solve the Schrödinger-equation we try to approximate the exact solution $U(t) = e^{-i\tau\mathcal{H}}$ by an approximation $\widetilde{U}(t)$.

The Suzuki-Trotter formula

$$U(t) = e^{-it\mathcal{H}} = e^{-it(\mathcal{H}_1+\ldots+\mathcal{H}_K)} = \lim_{m\to\infty}\left(\prod_{k=1}^{K} e^{-it\mathcal{H}_k/m}\right)^m \tag{11}$$

enables us to split up the operator $U(t)$ into time slices. This is mathematically exact for $m \to \infty$. For a numerical approximation we have to stop m at some point. The Baker-Hausdorff formula

$$e^{t(A+B)} = e^{tA}e^{tB}e^{t^2[A,B]}\ldots \tag{12}$$

allows us in the case $t\left\|\mathcal{H}_i\right\| \ll 1$ to rewrite the whole unitary propagator $U(t)$ as a product of a number of small (2x2) and (4x4)-matrices $U(t)_k$. So for sufficiently small t Suzuki-Trotter applied to 1st and 2nd order gives us

$$\widetilde{U}_1(t) = e^{-it\mathcal{H}_1}\ldots e^{-it\mathcal{H}_K} \tag{13}$$

$$\tag{14}$$

$$\widetilde{U}_2(t) = \widetilde{U}^\dagger_1(-t/2)\widetilde{U}_1(t/2) = e^{-it\mathcal{H}_K/2}\ldots e^{-it\mathcal{H}_1/2}e^{-it\mathcal{H}_1/2}\ldots e^{-it\mathcal{H}_K/2} \tag{15}$$

For the 2nd order approximation we can estimate the numerical error of the approximation to $\mathcal{O}(t^2)$.

Summarizing to compute the transition from $|\Psi(t)\rangle$ to $|\Psi(t+\tau)\rangle$ we decompose $U(t)$ into small time steps of length δ

$$U(t+\tau,t) = U(t+m\delta,t+(m-1)\delta)\ldots U(t+2\delta,t+\delta)U(t+\delta,t) \tag{16}$$

Then we use the 2nd order Suzuki-Trotter for each time step yielding

$$\widetilde{U}_2(t+m\delta, t+(m-1)\delta) = e^{-it\mathcal{H}_1(t+(m-\frac{1}{2})\delta/2} \dots e^{-it\mathcal{H}_K(t+(m-\frac{1}{2})\delta} \dots e^{-it\mathcal{H}_1(t+(m-\frac{1}{2})\delta/2} \quad (17)$$

where again $\mathcal{H}_k$ is a single term of the Hamiltonian

$$\mathcal{H}(t) = -J^z_{1,2}(t)S^z_1 S^z_2 - \sum_{i=1,2}\sum_{\alpha=x,y,z} h^\alpha_i(t)S^\alpha_i \quad (18)$$

Programming and Parallelisation

So far the program runs for 2 qubits and is able to perform all single qubit operations and different realisations of the CNOT gate. In the following a 2-qubit system is assumed with qubit 1 and 2 respectively.

Structure of the Program

Each elementary operation as for example the single-bit rotations is called a microinstruction (MI). In the program there are MIs for the rotations about the $x-, y-, z-$axis by $\frac{\pi}{2}$ and $-\frac{\pi}{2}$ for each qubit denoted by $X_i, \widetilde{X}_i, Y_i, \widetilde{Y}_i, Z_i, \widetilde{Z}_i$ with $i = 1, 2$ as well as an interaction operator I_{12} that can be interpreted as a rotation of one spin in the other's magnetic field. These operations are determined by the parameters $J^z_{1,2}$ and h^α_i in the Hamiltonian (eq.10). Considering a NMR quantum computer we have to keep in mind that the qubit-qubit interaction term has also to be applied during an external radio-pulse. Even though $J^z_{1,2}$ is usually much smaller than h^α_i the perturbation during the time of the radio-pulse is not that big. It is also important to consider that the radio pulse also influences not only the desired qubit to whose resonance frequency it is tuned but also with a smaller impact of course the other qubit. Therefore the $X-, \widetilde{X}-, Y-, \widetilde{Y}-$MIs consist of 2 operations: One that is applied to spin 1 and one that is applied to spin 2. The course of the program is as follows:

1. selection of the parameters h^α_i,$J^z_{1,2}$ and the step length δ (this defines the number of steps that it takes a MI to rotate the spin about the desired angle)
2. initialisation of the input state
3. for all successive operations being applied for the same time period
   - for each time step
     (a) compute the matrices for the next iteration step according to the Suzuki-Trotter decomposition and the time step length
     (b) call `qubit1ww` or `qubit2ww` (see next section)
4. readout of the expectation values

The Subroutines `qubit1ww` and `qubit2ww`

The core of the program are the subroutines `qubit1ww` and `qubit2ww` that do the actual application of the $(2\times 2)-$ and $(4\times 4)-$matrices to the state vector that is stored as an array containing the amplitudes of the basis states. The difficulty is that in order to apply the $(2\times 2)-$matrix in the case of a single-qubit operation to the correct pairs of amplitudes these have to be reordered. In this case the two amplitudes that belong together are the ones that have the same value for all qubits except the one the operation acts on. Therefore we use an auxiliary index array of the same length as the amplitude array initialised to the values $0 \dots 2^n$. Depending on which qubit the rotation is supposed to act on a bit-wise resorting algorithm

is used to bring the index array into the right order. Then the index array contains the order of the amplitudes in the amplitude array. Now the routine just takes the first two entries of the index array and uses these as indices to access the right amplitudes from the amplitude array, performs the multiplication and writes back the new amplitude values to the amplitude array. This procedure is shown in figure 3. Since especially for the 2-qubit-operations the matrices often have diagonal shape, the multiplication can be accelerated by implementing it manually.

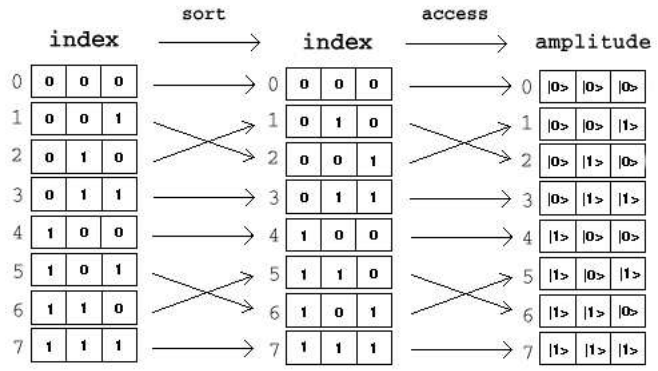


Figure 3: a single-qubit operation on the 2nd qubit of a 3-qubit system

OpenMP and MPI

The JUMP supercomputer consists of 40 nodes. Each of them has again 32 processors that have access to the same memory. OpenMP provides the programming commands for the communication between the processors of one node while MPI does the same for inter-node communication. The parallelisation process has been started with OpenMP because at the first glance there is one amplitude array on that all computations are carried out. So all processors that are participating in the computation have to access the same memory space. That makes OpenMP preferable as long as we use only one node or up to 32 processors respectively. For further parallelisation with MPI another strategy has to be thought of to integrate the already existing OpenMP code. The crucial point is that all MIs have to be applied successively and on a non-ideal quantum computer the outcome depends on the order of the applications. As a consequence parallelisation can only take place within one time step.

Coming back to the OpenMP parallelisation there are 2 major work steps that can be parallelised. On the one hand before every call of the subroutines `qubit1ww` and `qubit2ww` the actual matrices have to be computed. Thus while one processor is performing the actual computation another processor prepares the matrices for the next time step. On the other hand if the system consists of more than 2 qubits a partition of the index array and so a distribution of the amplitude array onto different processors makes sense having every processor just computing a part of the amplitude array. This is possible since within one MI acting on one qubit no further sorting is necessary and no confusion with the array entries can occur. This partition scheme is shown in figure 4.

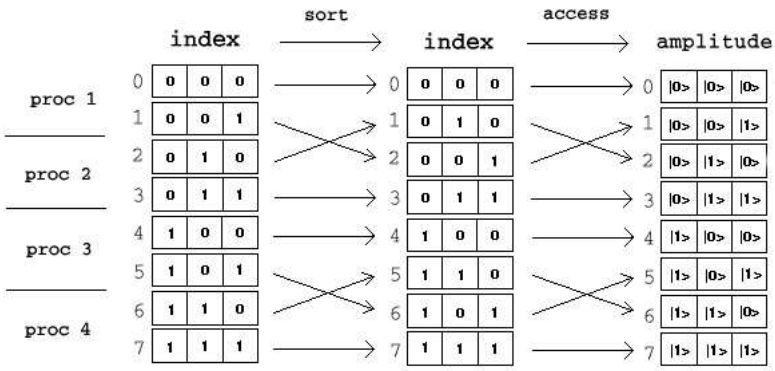


Figure 4: parallelisation in the case of 3 qubits

Memory Usage

To provide sufficient accuracy the variable type 'double complex' is used which occupies 2^4 bytes per complex number. Storing the $2^n \times 2^n$ matrix $U(t) = e^{-i\mathcal{H}t}$ of our $n-$qubit system would require 2^{2n+4} bytes. The state vector requires additional 2^{n+4} bytes of memory. Especially if we consider larger systems of about 20 qubits the needed memory size comes close to the available 5.2 Terabyte of the JUMP architecture. The decomposition of $U(t)$ into $(2 \times 2)-$ and $(4 \times 4)-$ matrices would allow us to simulate up to 38 qubits since now the state vector would take up most of the memory. In this case the memory consumption of the small operator matrices is negligible. This reduction of memory is of course at the expense of speed since every 2-qubit operation has to be applied 2^{n-1} times every 4-qubit 2^{n-2} times.

Conclusion and Outlook

Simulations of ideal problems give the correct results as it is easy to check with a pen and a paper. For the computation of different realisations of CNOT gates in the NMR case measuring the expectation values for the 2 qubits gives rudimentary the same results for the outcome states as published in [1] and [2]. The next step will be to implement other gates and algorithms as the Toffoli gate for example or the Grover's data search algorithm as well as to increase the number of qubits to be able to analyse decoherence phenomena.

Results

In order to test the program 3 different $CNOT-$gate implementations have been applied to the 4 basis states. The results seem to show the right tendency and the $CNOT_2$ implementation seems to be closest to the ideal $CNOT$-gate. Table 1 shows the expectation values $\langle Q_1^z \rangle$ and $\langle Q_2^z \rangle$ of qubit 1 and qubit 2. Here the differences between the ideal and the NMR-quantum computer become clearly apparent.

Acknowledgement

At this point I'd like to thank Dr. R. Esser for the coordination of the guest student program of the 'Zentralinstitut für Angewandte Mathematik des Forschungszentrums Jülich' as well as my two supervisors Dr. G. Arnold and Dr. M. Richter for their helpful support.

	Ideal quantum computer		$s=8$		$s=16$		$s=32$		$s=64$	
Operation	Q_1^z	Q_2^z	Q_1^z	Q_2^z	Q_1^z	Q_2^z	Q_1^z	Q_2^z	Q_1^z	Q_2^z
$(CNOT_1)\|00\rangle$	0.00	0.00	0.61	0.02	0.58	0.02	0.63	0.01	0.52	0.02
$(CNOT_2)\|00\rangle$	0.00	0.00	0.15	0.02	0.15	0.02	0.16	0.01	0.13	0.02
$(CNOT_3)\|00\rangle$	0.00	0.00	0.77	0.02	0.72	0.02	0.80	0.02	0.65	0.00
$(CNOT_1)\|01\rangle$	0.00	1.00	0.61	0.98	0.58	0.98	0.63	0.99	0.52	0.98
$(CNOT_2)\|01\rangle$	0.00	1.00	0.15	0.98	0.15	0.98	0.16	0.99	0.13	0.98
$(CNOT_3)\|01\rangle$	0.00	1.00	0.77	0.98	0.72	0.98	0.80	0.98	0.65	1.00
$(CNOT_1)\|10\rangle$	1.00	1.00	0.39	0.98	0.42	0.98	0.37	0.99	0.48	0.98
$(CNOT_2)\|10\rangle$	1.00	1.00	0.85	0.98	0.85	0.98	0.84	0.99	0.87	0.98
$(CNOT_3)\|10\rangle$	1.00	1.00	0.23	0.98	0.28	0.98	0.20	0.98	0.35	1.00
$(CNOT_1)\|11\rangle$	1.00	0.00	0.39	0.02	0.42	0.02	0.36	0.01	0.48	0.02
$(CNOT_2)\|11\rangle$	1.00	0.00	0.85	0.02	0.85	0.02	0.84	0.01	0.87	0.02
$(CNOT_3)\|11\rangle$	1.00	0.00	0.23	0.02	0.28	0.02	0.20	0.02	0.35	0.00

Table 1: expectation values for the differnt $CNOT-$operations, the parameter s can be seen as a measure of accuracy of the calculations

References

1. H. De Raedt, K. Michielsen, Computational Methods for Simulating Quantum Computers (2004)
2. H. De Raedt, K. Michielsen, A. Hans, S. Miyashita, K. Saito, Quantum spin dynamics as a model for quantum computer operation (2002)
3. M. A. Nielson, I. L. Chuang, Quantum Computation and Quantum Information Cambridge University Press (2000), ISBN 0-521-63503-9
4. J. Stolze, D. Suter, Quantum Computing - A Short Course from Theory to Experiment Wiley-Vch (2004), ISBN 3-527-40438-4

Fast Parallel Matrix Multiplication
Strassen - Winograd Algorithm

Armin Rund

Universität Bayreuth
Fakultät für Mathematik und Physik

E-mail: armin.rund@gut-grunau.de

Abstract:

This article describes different approaches to speed up matrix-matrix multiplications. In the serial case three algorithms are compared. An optimized assembler code is combined with the serial and parallel algorithms. A concept for parallel matrix-matrix multiplication is discussed.

Introduction

Multiplication of two matrices is one of the most basic operations in scientific computing. Its central role is emphasised by its inclusion in portable libraries, such as the Level 3 BLAS. A speed up of this basic operation would increase the performance of every application using matrix-matrix multiplications.

There are two possibilities to reduce the runtime of the computation. Tuning the classical algorithm to a given machine architecture, and applying alternative algorithms with asymptotic complexity less than the $\mathcal{O}\left(m^3\right)$ operations required by the classical algorithm.

This paper addresses speeding up the matrix-matrix multiplication of dense matrices using both approaches. The results show that both an alternative algorithm and an adjustment of the existing algorithm to the architectures can accelerate the computation and indicate how these approaches may be combined.

Background

This section gives a short overview of the libraries, programming language, and the architecture, that were related to the work.

JUMP

The JUMP (JUelich Multi Processor) is a distributed shared memory parallel computer situated at the ZAM, Forschungszentrum Jülich. It consists of 41 nodes each containing 32 IBM POWER4+ processors running at 1.7 GHz. Each node has a shared 128 GB main memory and a 3-step cache hierarchy where the most important cache level is the L2 cache with 1.5 MB per chip (2 processors) and an access time of 10-12 cycles. All in all, there are 1312 processors and an aggregate peak performance of 8.9 TFLOPS.

MPI

MPI (Message Passing Interface) is a library of functions and macros that can be used in C, FORTRAN and C++ programs. It is intended for use in programs that make use of multiple processors on distributed memory machines by message passing. It is one of the first standards for programming parallel processors.

BLACS

The BLACS (Basic Linear Algebra Communication Subprograms) are a linear algebra oriented message passing interface. It is implemented for a variety of hardware architectures. The interface is identical across the different distributed memory platforms and independent of the message passing libraries, which makes it easy to develop portable applications for linear algebra problems. On the JUMP, the BLACS are built upon the IBM communication interface LAPI (Low-level Application Programming Interface). The BLACS are used as communication layer for the ScaLAPACK project and for the PESSL.

BLAS

The BLAS (Basic Linear Algebra Subprograms) are high quality routines for performing basic vector and matrix operations. Optimised versions of this library exist for almost all systems. Thus programs that are based on the BLAS perform very well on different architectures. The BLAS are organised in 3 parts. The Level 1 BLAS perform vector-vector operations like vector addition or copy operations. The Level 2 BLAS do matrix-vector operations like the matrix-vector product or rank-one updates of a matrix. The Level 3 BLAS do matrix-matrix operations, among other things the matrix-matrix multiplication routine `DGEMM`. It is a very efficient algorithm for matrix multiplication and is thus widely used. BLAS are included in ESSL.

PBLAS

The Parallel BLAS routines are distributed-memory versions of the BLAS. The PBLAS use the BLACS for communication between processes and the BLAS for computation in a process. The PBLAS are written in C, but with Fortran 77 interfaces. Both, ScaLAPACK and PESSL, contain the PBLAS.

GOTO

GOTO are high-performance BLAS routines provided by Kazushige Goto [6]. The routines are written in assembler code. They achieve better performance on current generation architectures by reducing the overhead originating from Translation Look-aside Buffer (TLB) table misses. The library is available for about 20 different architectures and can be accessed via the internet [5]. Especially, GOTO contains an optimised version of DGEMM for matrix multiplication. It replaces the BLAS routine by linking it prior to or instead of ESSL.

FORTRAN

The software developed is written in FORTRAN 90. Compared to older versions of FORTRAN, it offers modern language features like dynamic memory allocation or the module concept. The FORTRAN 90 standard was designed to exist along with older versions of FORTRAN, although the cooperation is not always easy to handle. Since the PBLAS are written in C, also some C files exist.

Serial General Matrix Multiplication

Consider the problem of computing a product of two matrices:

$$A : m \times k \quad \text{and} \quad B : k \times n$$

The first attempt to program a matrix multiplication makes use of three loops (Figure 1).

```
DO i=1,m
      DO j=1,n
            DO  l=1,k
                  C(i,j) + =  A(i,l)*B(l,j)
            ENDDO
      ENDDO
ENDDO
```

Figure 1: Matrix multiplication with 3 loops

Adaptation to the Architecture

Since current computers have a hierarchical memory structure in which the access to data in upper levels (registers, cache) is faster than to data in lower levels, it is a common technique to use the data stored in the upper levels as often as possible. Therefore, the matrices are partitioned into blocks and the computations are then performed on the blocks generated.

The computation deals with $m \cdot k \cdot n$ multiplications and additions. Since the matrices A, B, C have to be stored, memory for $m \cdot k + k \cdot n + m \cdot n$ numbers is needed. Assuming quadratic matrices and treating an addition together with a multiplication as an operation, we obtain m^3 operations and a storage of $3m^2$.

As the dimension m increases, the matrices exceed the cache of the architecture leading to an accumulation of cache misses. To avoid this, a blocked algorithm is applied. The matrices are recursively split up into parts, until the resulting parts suit the cache size. One level of this splitting process bisectioning the dimensions is illustrated in Figure 2.

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Figure 2: Classical matrix multiplication

A set of 8 independent matrix multiplications are created. These need only a fourth of the memory compared to the original matrix multiplication. Recursively, the matrix size decreases, thereby allowing the matrices to fit into the cache.

The number of operations remains the same. Nevertheless, the computation is sped up, as the number of cache misses drops dramatically.

Splitting further than to blocks that fit into L2 cache is counter-productive, so an optimal crossover point can be computed for every architecture, above which a partitioning is more effective than computation at that level.

$$C \Leftarrow \alpha\, op(A) \cdot op(B) + \beta\, C$$
$$\text{with } op(X) = X \text{ or } X^T$$
$$\alpha,\ \beta \in \mathbb{R}$$

Figure 3: Operation of DGEMM

The Level 3 BLAS routine `DGEMM` makes use of such a cache optimization. It performs the operation pictured in Figure 3. To further accelerate the computation of a general matrix-matrix multiplication, the number of operations has to be reduced. There are some algorithms offering a lower amount of (computational) complexity. A class of them is discussed in the next section.

Applying an Alternative Algorithm

One algorithm which offers an asymptotic complexity less than the classical algorithm is Strassen's algorithm. It was introduced in 1969 ([7]).

The idea of Strassen's algorithm is to avoid 1 of the 8 multiplications that are generated in the partitioning depicted in Figure 2. The procedure is divided into 3 phases, the pre-additions, the multiplications and the post-additions (Figure 4).

The standard algorithm that performs $\mathcal{O}(m^3)$ is shown in Figure 2. Strassen's algorithm uses algebraic identities to reduce the number of multiplications to 7 at a cost of 18 additions instead of 4. Recursion leads to a drop of the complexity down to $\mathcal{O}\ (m^{2.807})$.

Note that asymptotically (m » 18), the matrix additions can be neglected, because of complexity of $\mathcal{O}(m^2)$ compared to the complexity of matrix multiplication ($\mathcal{O}(m^3)$). The additions do cause a higher amount of memory accesses. This plays a central role in designing a parallel matrix multiplication algorithm.

Winograd's variant of Strassen's algorithm ([1]) uses 7 recursive matrix multiplications and 15 additions. It is depicted in Figure 5. It can be shown that this is the minimum number of multiplications and additions for any recursive matrix multiplication based on a partition into quadrants ([2]). The number of additions compared to Strassen's algorithm is reduced at the cost of a slightly further increased number of memory accesses. For a serial implementation memory accesses can be cheap, if the sequence of the operations is arranged properly. Thus Winograd's algorithm is preferable in this case.

Both algorithms outperform the classical algorithm above a critical matrix dimension, referred to as

mindim. Below that dimension the classical algorithm is preferable. It is a characteristic value of the architecture. C. Douglas et. al. ([1]) recommend *mindim* to be between 32 and 256. For current systems it should be somewhat higher. A suitable crossover point for JUMP is between 800 and 1650.

$$
\begin{array}{lll}
S_1 = A_{11} + A_{22} & M_1 = S_1 S_6 & T_1 = M_1 + M_4 \\
S_2 = A_{21} + A_{22} & M_2 = S_2 B_{11} & T_2 = M_7 - M_5 \\
S_3 = A_{11} + A_{12} & M_3 = A_{11} S_7 & T_3 = M_1 + M_3 \\
S_4 = A_{21} - A_{11} & M_4 = A_{22} S_8 & T_4 = M_6 - M_2 \\
S_5 = A_{12} - B_{22} & M_5 = S_3 B_{22} & \\
S_6 = B_{11} + B_{22} & M_6 = S_4 S_9 & \\
S_7 = B_{12} - B_{22} & M_7 = S_5 S_{10} & C_{11} = T_1 + T_2 \\
S_8 = B_{21} - B_{11} & & C_{12} = M_3 + M_5 \\
S_9 = B_{11} + B_{12} & & C_{21} = M_2 + M_4 \\
S_{10} = B_{21} + B_{22} & & C_{22} = T_3 + T_4
\end{array}
$$

Figure 4: Strassen's algorithm

$$
\begin{array}{lll}
S_1 = A_{21} + A_{22} & M_1 = S_2 S_6 & T_1 = M_1 + M_2 \\
S_2 = S_1 - A_{11} & M_2 = A_{11} B_{11} & T_2 = T_1 + M_4 \\
S_3 = A_{11} - A_{21} & M_3 = A_{12} B_{21} & \\
S_4 = A_{12} - S_2 & M_4 = S_3 S_7 & \\
S_5 = B_{12} - B_{11} & M_5 = S_1 S_5 & C_{11} = M_2 + M_3 \\
S_6 = B_{22} - S_5 & M_6 = S_4 B_{22} & C_{12} = T_1 + M_5 + M_6 \\
S_7 = B_{22} - B_{12} & M_7 = A_{22} S_8 & C_{21} = T_2 - M_7 \\
S_8 = S_6 - B_{21} & & C_{22} = T_2 + M_5
\end{array}
$$

Figure 5: Winograd's variant of Strassen's algorithm

Implementation

A serial Winograd algorithm was coded in FORTRAN 90. The routine is named `DGEMMW`. It has the same functionality and parameters as the Level 3 BLAS routine `DGEMM`. It executes the operation depicted in Figure 3 with matrices containing values in `DOUBLE PRECISION`.

The algorithm is consistent with [1]. There are two features to point out:

At first, one has to take care of dealing with odd dimensions. As pointed out in [4], there are several possibilities to solve the problem: by embedding the matrix into a larger one (*static padding*), by cutting off some rows / columns at each level and compute their contribution to the outcome separately (*dynamic peeling*) or by decomposing the matrix into parts that overlap by one row / column at each level (*dynamic overlap*). Here dynamic overlap is used. Note, that the duplication of rows or columns is only conceptionally. There is almost no extra storage or extra computation. In FORTRAN 90 one only has to select the right submatrices, then the whole algorithm requires only 10 extra lines of code for handling odd sized matrices. For details on dynamic overlap, refer to [1].

The second feature to emphasise is the low amount of extra storage required in an implementation according to [1]. The operations are arranged in a special order to maximise the reuse of auxiliary storage. For quadratic matrices, the auxiliary storage is $2/3m^2$ in the case $\beta \neq 0$, otherwise $5/3m^2$ because the matrix C has to be stored. In the last case, the amount of storage could further be reduced ([3]) at the expense of extra arithmetic operations and a more complex implementation. It is not further covered here.

The performance of DGEMMW is depicted in Figure 7. To gain further performance, DGEMMW is combined with the cache-optimised assembler code of K. Goto ([6]). See the results in Figure 8.

Parallel General Matrix Multiplication

In this chapter, a concept for parallel matrix-matrix multiplication is discussed.

Array Layouts

Parallel algorithms require that the global data is distributed across the processes prior to computation. The layout of the data is critical for the performance of parallel code. There are two main issues to consider in choosing a data layout for matrix-matrix computations. On the one hand, the load balance is important. That means the work should be evenly distributed among the processes thus reducing the idle waiting time of the processors. On the other hand, the amount of communication affects the efficiency of the algorithm. Since the distribution should allow an application of BLAS routines at the lowest level, communication arises throughout the parallel computations and especially prior to the call of BLAS routines. A minimal amount of message passing throughout the algorithm is the second main objective.

Block Cyclic Data Distribution

The block cyclic data distribution provides a simple and flexible way of distributing a block-partitioned matrix on distributed memory machines. It is the recommended distribution scheme for dense matrix operations with the PBLAS ([8]). The p processes are ordered in a 2-dimensional rectangular $p_r \times p_c$ grid with $p = p_r \cdot p_c$. The matrices are partitioned into equal blocks of blocksize $MB \times KB$ beginning from the upper left. If the dimensions can not be divided by the blocksize, the rightmost column or the lowest row might consist of fragmentary blocks. Afterwards, the blocks are distributed on the grid, starting again from the upper left.

Figure 6 shows an example of a block cyclic data distribution on a rectangular grid of 2×3 processes. The processes are numbered from 0 to 5. All blocks labeled with the same number belong to the same process.

In general, this distribution allows an reasonable load balance for a great variety of problems, e.g. for the LU-decomposition, especially if $MB \cdot p_r < m$. Nevertheless there are distributions with better load balance, for instance imagine the last row in Figure 6 distributed on $0|1|2|3|4|5$.

Moreover, block cyclic data distribution is very flexible. It can reproduce most data distributions that are used in linear algebra computations like the column block distribution or the column cyclic distribution.

$$\left(\begin{array}{c|c|c|c|c|c} 0 & 1 & 2 & 0 & 1 & 2 \\ \hline 3 & 4 & 5 & 3 & 4 & 5 \\ \hline 0 & 1 & 2 & 0 & 1 & 2 \\ \hline 3 & 4 & 5 & 3 & 4 & 5 \\ \hline 0 & 1 & 2 & 0 & 1 & 2 \end{array}\right)$$

Figure 6: Block cyclic data distribution

Other Array Layouts

Another approach for improving both load balance and locality is the use of quadtrees (*Morton ordering*) or space filling curves (*Hilbert ordering*). According to [2] recursive array layouts outperform tradi-

tional layouts for the standard algorithm. For Strassen's and Winograd's algorithm they offer only little improvement.

Parallel Algorithms

There are two possibilities to take advantage of the reduced complexity that Strassen's or Winograd's algorithm provide: The serial level and the parallel level.

Applying Winograd in the Serial Level

A serial version of Winograd can substitute the `DGEMM` above the crossover point. Because all of the data is locally available, the larger number of memory accesses can be compensated for by clever organisation. Thus Winograd's algorithm outperforms `DGEMM` and Strassen's algorithm, as it offers the lowest number of operations. Adapting the crossover point optimally will speed up each serial computation and thereby accelerate the parallel matrix multiplication.

Note, that the matrices passed from the parallel algorithm to the sequential Winograd should be nearly quadratic. If one of the dimensions is very small compared to the others, the Winograd algorithm will do only few iterations or in the worst case no iterations at all. In this case, the algorithm is slowed down.

Applying Winograd in the Parallel Level

Strassen's or Winograd's algorithm can also be applied in the parallel case. Unfortunately, the increase in memory accesses can in general not be negleted as in the serial case. According to [2] the algorithmic locality of reference is much worse in case of Strassen's and Winograd's algorithms than in case of the standard algorithm. Thus the benefits of the reduced complexity are offset by the larger number of memory accesses, which increases the amount of message passing. It is dependent on the architecture, more precisely on the ratio of communication latency to local memory latency, whether there is a speed-up or a slow-down compared to applying the classical blocking algorithm in the parallel part.

S. Chatterjee et. al. ([2]) observed, that there is indeed no advantage of Winograd's algorithm compared to Strassen's algorithm, as Winograd's loss of algorithmic locality compensates for its advantage of lower complexity. Furthermore, Strassen's algorithm shows a better possibility to parallelize the code. Compare Figure 4 to Figure 5: In case of a Winograd's variant, the pre- and post-additions are serial leading to a limited possibility of parallelisation as some processes have to wait for others finishing their additions. In case of Strassen's algorithm, all pre- and post-additions could be executed simultaneously. Hence, Strassen's algorithm is preferable in the parallel case.

However, the choice between Strassen's algorithm and the classical algorithm also depends on some other factors. A parallel Strassen's algorithm has a high dependency on the number of processes. Remember that the number of generated multiplications is a power of 7, so the load balance is quite difficult to handle. Moreover, the array layout plays an important role. A general array layout like the block cyclic distribution generates a poor locality of the data needed in the pre-additions and post-additions, leading to a loss of speed through additional traffic.

As the performance of a parallel Strassen's algorithm is not predictable in general, the first choice is to combine a parallel classical algorithm with a serial Winograd's algorithm.

Hybrid Algorithm

In the following, a hybrid algorithm based on serial Winograd combined with a classical parallel algorithm is exposed.

The first attempt is to connect the widely spread PBLAS routine `PDGEMM` with a serial Winograd. Unfortunately, this concept does not succeed. As mentioned above, the serial Winograd requires nearly quadratic matrices for a speed-up. In contrast, `PDGEMM` works with a block cyclic distribution where, in the lowest parallel level, each process gets several small matrix multiplications with by far not quadratic dimensions, because `PDGEMM` mainly partitions the column-dimension of A and the row-dimension of B. This dimension is cut down to the blocksize. Since the resulting size is usually smaller than *mindim*, the sequential Winograd algorithm will immediately call the `DGEMM` routine, just generating organisational overhead.

For example, `DGEMMW` was combined with `PDGEMM` from ScaLAPACK (since the code is open). A matrix multiplication of quadratic matrices with $m = 4000$ on a grid of 1×2 processes created on each process 40 calls to the serial Winograd containing matrix multiplications with the dimensions $m = 2000$, $k = 100$, $n = 4000$. Therefore, its run-time of 30 sec. compared to 14 sec. of the conventional `PDGEMM` is no surprise.

To embed a sequential Winograd effectively into a parallel routine, each process should compute only one big and nearly quadratic matrix multiplication. Besides, the parallel routine has to fulfil the following conditions:

- Data distribution should be compatible to the other routines working on that data
- Overall communication should be minimal
- Overall amount of memory should be minimal

C. Douglas et. al ([1]) offer a possible combination of a parallel algorithm performing $\mathcal{O}\ (m^3)$ with a serial Winograd. The parallel algorithm is based on a partitioning according to the prime factors of the number of processes. Assuming the number of processes to have some prime factors, this partitioning has the following advantages:

- Very good load balance
- Best condition for applying a Winograd algorithm (1 nearly quadratic multiplication)
- Low memory demand

The disadvantages are the additional communication compared to `PDGEMM` and the unconventional distribution of the matrices. Prior to and after the computation, a conversion to a common data distribution like the block cyclic will be necessary.

In [1] this hybrid algorithm shows better results than a classical parallel implementation.

By the way, this algorithm provides a starting point for a combination of OpenMP with MPI, since a group of processes collaborates on each part of the matrices.

Tuning the Existing Algorithm

As an alternative to introducing new algorithms, the existing PBLAS routine `PDGEMM` could be modified. This approach involves the advantage of working with a compatible data distribution (block cyclic distribution).

As explained above, the efforts of combining PDGEMM with a serial Winograd's algorithm failed due to the unbalanced partitioning of PDGEMM. In another attempt, the call to DGEMM (ESSL) in the lower level is linked to the GOTO assembler code. According to Figure 8 the assembler code outperforms DGEMM. In the parallel case, there is also a gain in performance (Figure 10).

Performance Results

For performance analysis the time is measured using the FORTRAN 95 intrinsic subroutine CPU_TIME. The matrices are assumed as quadratic. The figures show the time of computation plotted against the dimension m of the matrices.

DGEMMW

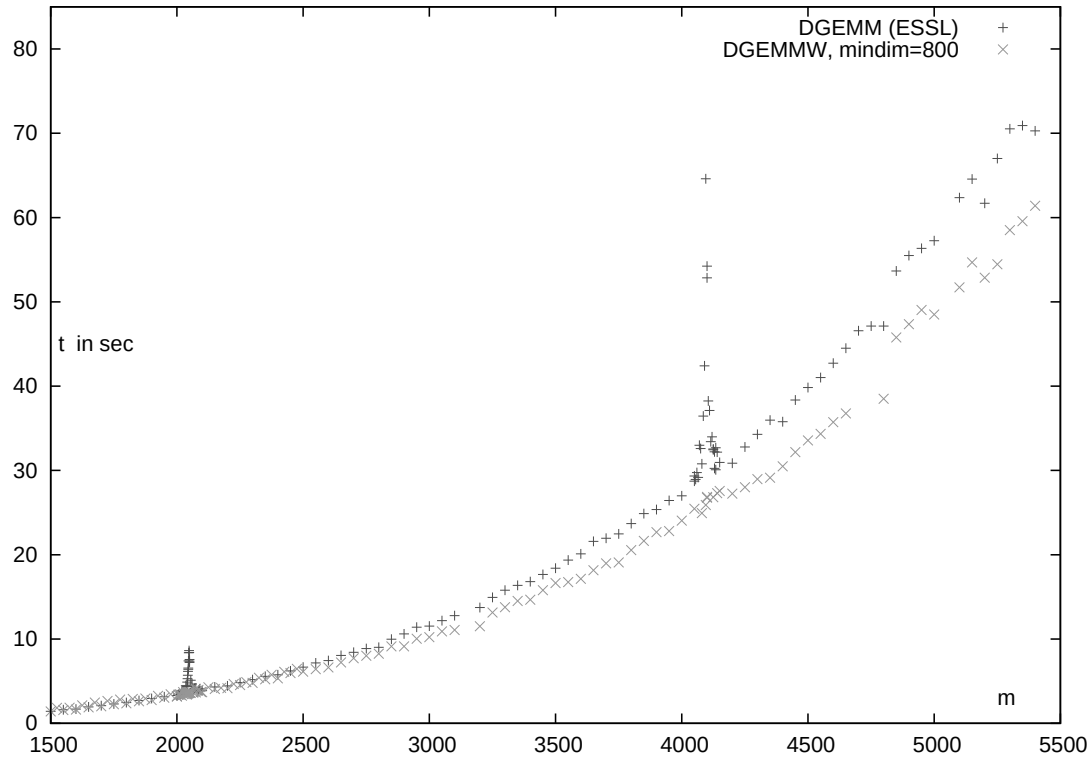


Figure 7: Comparison of DGEMM and DGEMMW

Figure 7 shows the run duration of the level 3 BLAS DGEMM taken from the ESSL in comparison to DGEMMW. As crossover point, $mindim = 800$ is chosen. DGEMMW outperforms DGEMM above a dimension of 3200, where at least 3 levels of Winograd's algorithm are carried out. The two peaks in the upper curve are due to the known performance breakdown of DGEMM as soon as the dimensions approximate powers of 2. In contrast, DGEMMW shows no striking breakdown at these points, as the dimensions passed to DGEMM at the crossover point are always around 512 independent of the actual exponent of 2. Since DGEMM gets on with that small dimension, the high peak is not transfered to DGEMMW.

In Figure 8, the middle curve describes the runtime of the assembler code GOTO. There is also no significant performance breakdown at certain dimensions. The lowest curve shows the performance of combining the serial Winograd with the assembler code. Compared to DGEMM ESSL, there is a speed-up of up to 20% at dimensions around 4500. As the asymptotic complexity of Winograd is lower, this percentage will generally rise with the dimension.

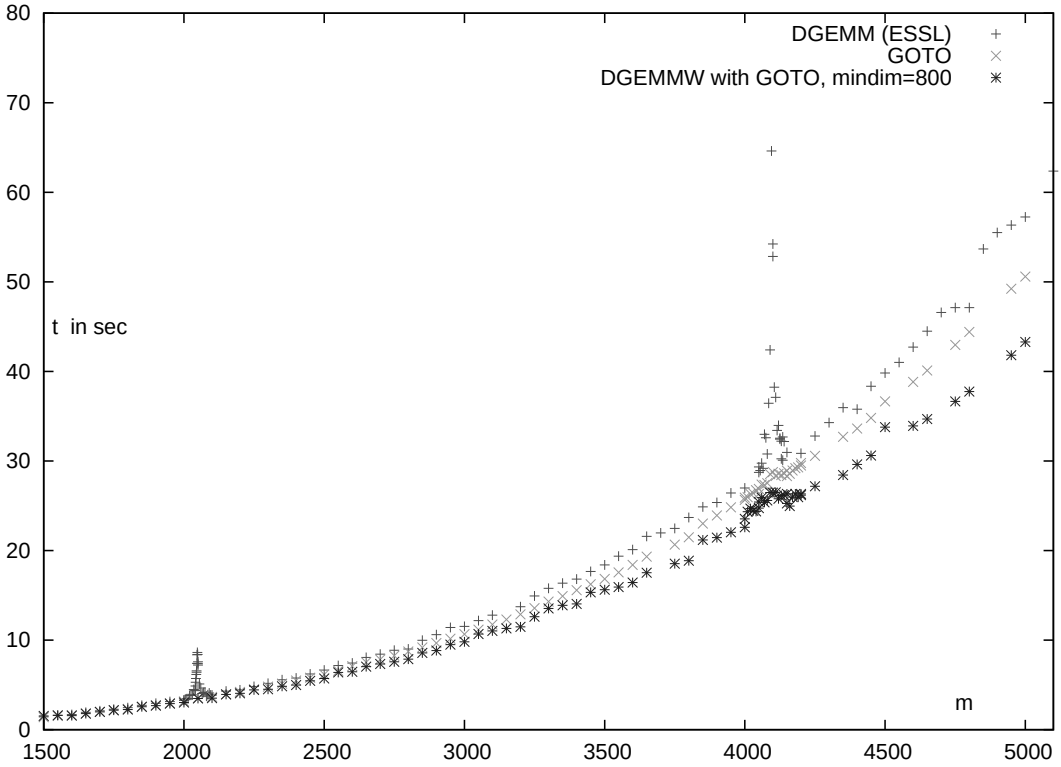


Figure 8: Comparison of DGEMM, GOTO and DGEMMW + GOTO

PDGEMM

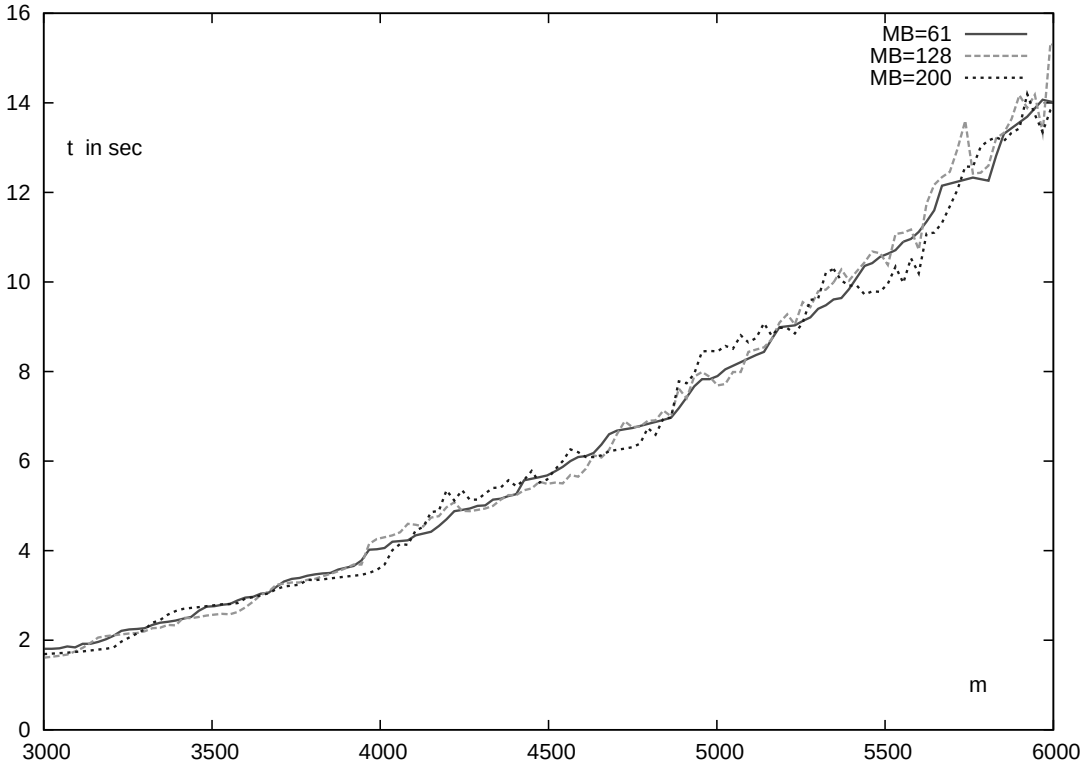


Figure 9: PDGEMM PESSL for different blocksizes

Because of the block cyclic distribution, the performance of `PDGEMM` depends on the blocksize. So at first one has to choose a suitable blocksize. Figure 9 illustrates the performance of `PDGEMM` in PESSL for 3 different blocksizes. The curves oscillate around each other, since the load balance changes periodically. Increasing the matrix size with a fixed blocksize causes the block-cyclic data distribution to vary from the best case to the worst case, where only one row of processes gets a full additional row of blocks. Thereby, there is an optimal blocksize for each matrix dimension (the lowest curve). In contrast, if the matrix dimensions to deal with are not given in advance, it does not make a serious difference which blocksize is chosen, since on average all 3 plots show the same behaviour. In this case, the blocksize can be chosen suitable for other routines working with the distribution.

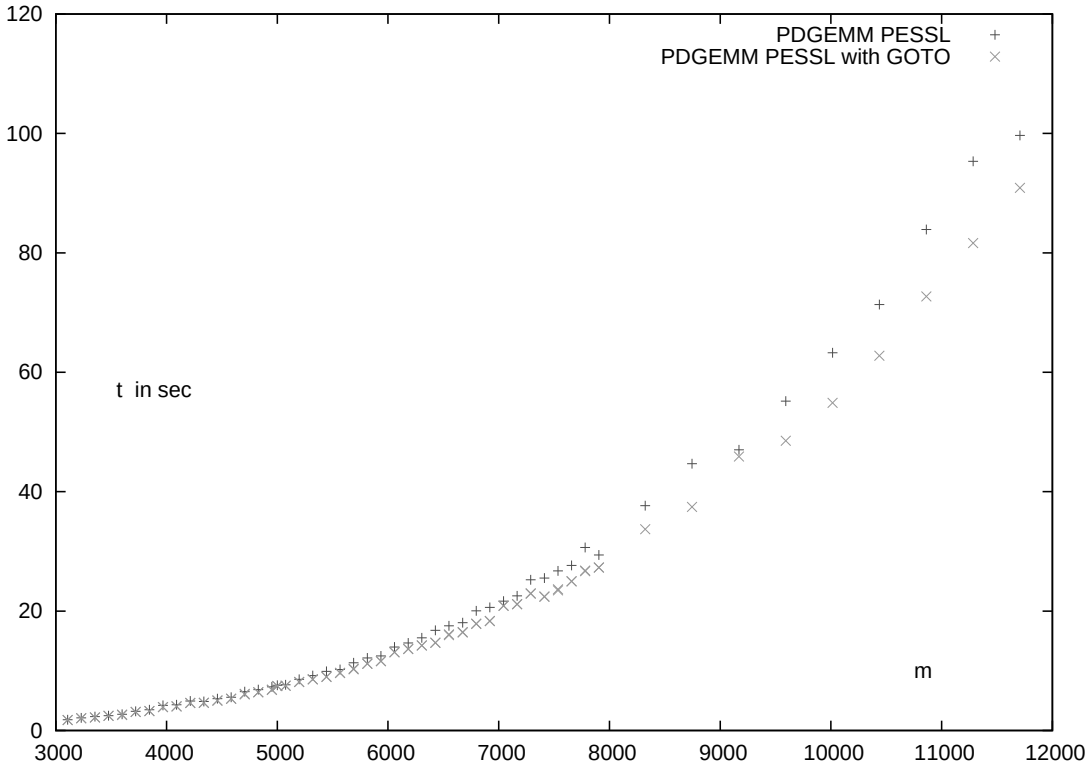


Figure 10: PDGEMM PESSL with GOTO

Figure 10 illustrates the connection between the existing routine `PDGEMM` (PESSL) and the assembler code. As expected, GOTO accelerates the computation by a certain rate.

Conclusion and Outlook

Alternative algorithms can bring a certain speed-up in matrix-matrix multiplication, especially considering problems with great complexity. However, the application of alternative algorithms is not generally preferable as they also bring disadvantages.

Applying cache optimisation techniques on algorithms allows further speed-up. It is a surprising result that a public domain assembler code can even outperform the optimised IBM code.

The next step would be an implementation of the hybrid algorithm described above. It is expected to be a competitive alternative to `PDGEMM`.

An application of Strassen's algorithm in the parallel level is also possible, but the implementation would be quite difficult and the outcome unpredictable.

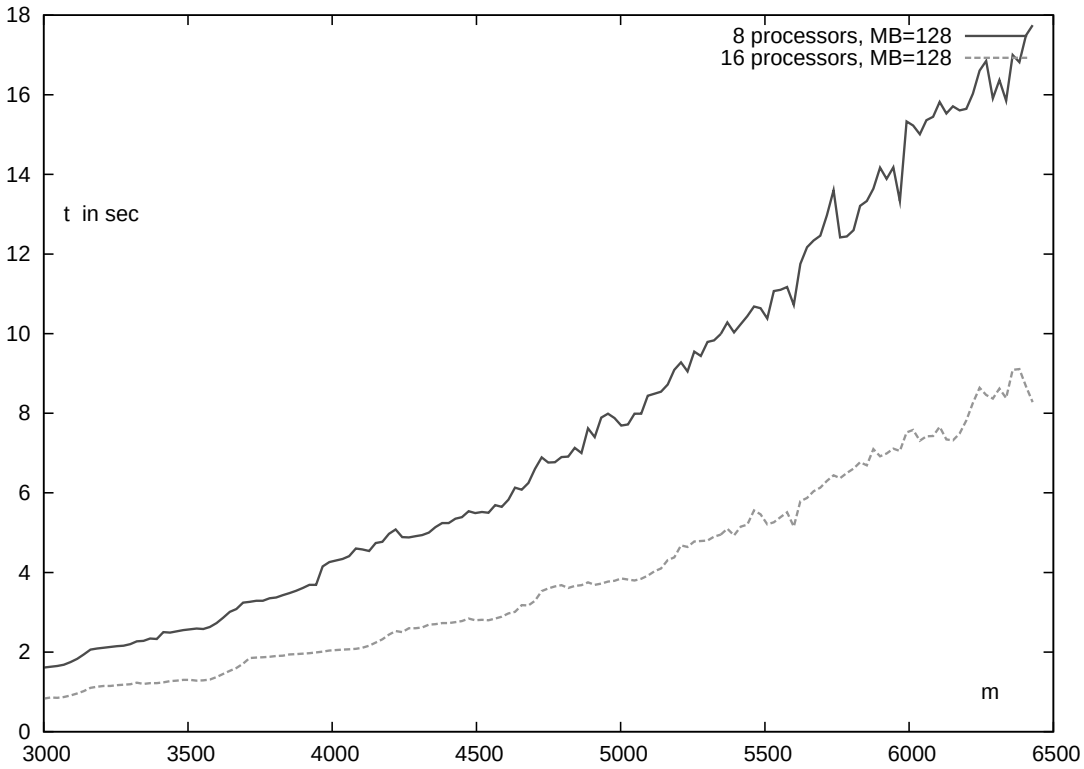


Figure 11: PDGEMM PESSL for different number of processes

Acknowledgements

I would like to thank my advisor Dr. Bernhard Steffen for supporting me during my stay at the Research Centre Jülich, Inge Gutheil for her help whenever I had problems with the existing software and Dr. Rüdiger Esser for organising the guest student program of the ZAM. These persons made my stay here a rich and precious experience.

References

1. C. Douglas, M. Heroux, G. Slishman, R. M. Smith. *GEMMW: A portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm*. **Journal of Computational Physic**, 110:1-10, 1994
2. S. Chatterjee, A. R. Lebeck, P. K. Patnala, M. Thottethodi. *Recursive array layouts and fast parallel matrix multiplication*. **Proc. 11-th ACM SIGPLAN**, June 1999
3. S. Huss-Lederman, E. M. Jacobson, J. R. Johnson, A. Tsao, T. Turnbull. *Strassen's algorithm for matrix multiplication: modelling, analysis, and implementation.* **Technical Report CCS-TR-96-147, Center for Computing Sciences**,1996
4. M. Thottethodi, S. Chatterjee, A. R. Lebeck. *Tuning Strassen's matrix multiplication for memory efficiency*. **Proceedings of SC98**, Orlando, Nov. 1998
5. High-Performance BLAS by Kazushige Goto, *http://www.cs.utexas.edu/users/flame/goto/*
6. K. Goto, R. van de Geijn. *On reducing TLB misses in matrix multiplication*. **The University of Texas at Austin, Department of Computer Sciences**, FLAME Working Note 9, Technical Report TR-2002-55, Nov. 2002
7. V. Strassen. *Gaussian elimination is not optimal*. **Numer. Math.**, 13:354-356, 1969
8. L. S. Blackford et. al. *ScaLAPACK Users' Guide*. **SIAM**, Philadelphia, 1997

Visualisierung von MHD-Daten mit Virtual-Reality-Techniken

Sebastian Schiffner

Freie Universität Berlin
Fachbereich Physik

E-Mail: seeh1@web.de

Zusammenfassung:

Ziel der Arbeit war die Erweiterung eines bestehenden OpenGL Programms zur Visualisierung von Magneto-Hydro-Dynamischen Vorgängen im Erdinneren. Realisiert wurden eine flexible Isolinien-Darstellung sowie die Visualisierung von Isoflächen, welche mittels des Marching-Cube-Algorithmus gewonnen werden. Zur interaktiven 3D-Steuerung wurde das Trackingsystem 3Space$^{(R)}$ Fastrak$^{(R)}$ von Polhemus in das Programm integriert. Ferner wurde eine Stereo-Darstellung der Anzeige durch die Verwendung von zwei Framebuffern implementiert. Somit wurde erreicht, dass das Visualisierungsprogramm auch in Virtual-Reality-Systemen zum Einsatz kommen kann.

Einführung

Das Programm bezieht seine Eingabedaten aus einer Binär-Datei, welche durch Computersimulation im Zentralinstitut für Angewandte Mathematik (ZAM), Forschungszentrum Jülich, gewonnen wurde. Die Grundlagen der Magneto-Hydro-Dynamik bauen darauf auf, dass schnell strömende heiße Gase und Flüssigkeiten unter Einfluss eines äußeren Magnetfeldes eine Spannung induzieren. Durch Konvektionsströme aufgrund der Erdrotation und Gravitation entsteht das Erdmagnetfeld, denn ein stromdurchflossener Leiter erzeugt nach der 'Rechten-Hand-Regel' ein Magnetfeld. Die Simulation des Erdmagnetfeldes beruht auf den Gleichungen der MHD, welche im übrigen ein Zusammenspiel aus Elektro- und Hydro-Dynamik sind. Die Simulation geschah in einer Zeitskala von ca. 1 Million Jahren (ungefähr 2000 Zeit-Schritte zu je 500 Jahren), vgl. [1]. In der Datei ist neben dem Magnetfeld und der Strömung auch noch die Temperatur der Erde gespeichert. Dabei ist das Gitter relativ klein, je nach Datentyp ungefähr $15 \times 15 \times 15$ Werte.

Abbildung 1 vermittelt einen ersten Eindruck, wie das Programm die MHD-Daten darstellt. Die Grundfunktionen der Software waren schon vor Beginn dieser Arbeit voll implementiert. Das rechte Bild in Abbildung 1 zeigt die Verwendung der Option 'contouring', mit der Isolinien eingezeichnet werden. Dabei ist zum Zeitpunkt der Aufnahme schon der Wert für die zweite (weiße) Linie frei wählbar. Zusätzlich zur Erzeugung der Isolinien wurde das Visualisierungsprogramm um die Möglichkeit der Darstellung von Isoflächen erweitert. Dies wird im Abschnitt Visualisierung von Isoflächen näher erklärt. Danach kommen Erläuterungen zum verwendeten Trackingsystem. Es wird darauf eingegangen, was beim Einsatz solcher Hardware beachtet werden muss. Da die Software das Bild stereoskopisch darstellen sollte, mussten noch zusätzlich einige OpenGL Befehle eingebaut werden. Welche genau das waren und wie sie funktionieren wird im Absatz OpenGL Feinheiten beschrieben. Den Abschluss bildet ein Ausblick über noch weitere denkbare und gewünschte Funktionen.

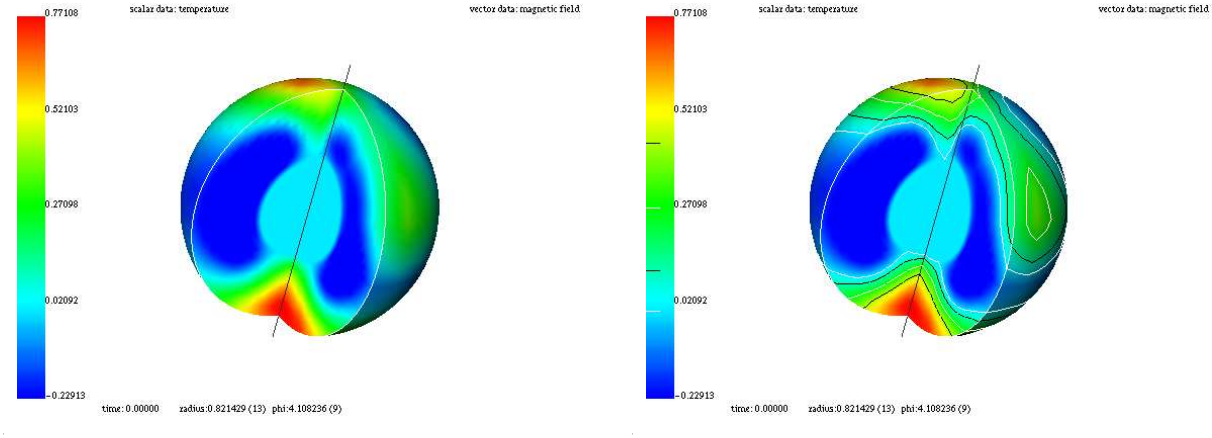


Abbildung 1: Links Temperatur der Erde, rechts daneben mit aktiviertem Contour-Plot

Visualisierung von Isoflächen

Es gibt mehrere Arten, wie sich Isoflächen finden und darstellen lassen. Denkbar ist zum Beispiel, dass die Funktion, die es darzustellen gilt, explizit gegeben ist. Dies ist in unserem Fall nicht so, weswegen die Daten nach solchen Flächen erst durchsucht werden müssen. Ein Algorithmus, der dies erledigt und mit dem man die gefundene Fläche auch gleich darstellen kann, ist der Marching-Cube-Algorithmus. Der Algorithmus ermöglicht es, aus skalaren Volumen-Daten Flächen gleichen Wertes, also Isoflächen, zu erzeugen. Hierbei wird ein Dreiecksnetz erzeugt, welches die gesuchte Isofläche approximiert. Anschaulich kann man sich das Vorgehen so vorstellen:

- Nehme einen Würfel und setze ihn an den Anfang des Volumendatensatzes
- Immer dann, wenn der Daten-Wert an einer der Ecken des Würfels über und an einer anderen Ecke unter dem gesuchten Schwellenwert liegt, schneidet die gesuchte Isofläche die Kante zwischen diesen Ecken. In diesem Fall wird auf der Kante ein Eckpunkt eines Dreiecks gelegt.
- Überprüfe dies für alle Ecken des Würfels
- Setze den Würfel im Volumen-Datensatz eine Position weiter und wandere so durch den Datensatz

Es gibt eine ganze Menge an Feinheiten zu dem Algorithmus. Es gibt zum Beispiel $2^8 = 256$ Möglichkeiten, wie der Würfel geschnitten werden kann. Daraus ergeben sich nach Lorenson und Cline [2] aber nur fünfzehn nichtreduzierbare Fälle. Aus diesen kann man durch Rotation, Spiegelung und Komplementär-Bildung alle anderen erzeugen. Als Beispiel überlege man sich, dass es keinen Unterschied macht, ob alle acht Eckpunkte unter dem gesuchten Schwellenwert oder darüber liegen, in beiden Fällen wird keine Kante geschnitten, dies entspricht Fall 1 in Abbildung 2.

Programmiertechnisch werden diese Fälle und die daraus resultierenden Schnittpunkte in einer so genannten Lookup-Tabelle gespeichert. Die Tabelle enthält sämtliche resultierenden Schnittpunkte für alle 256 Fälle. Welcher Fall vorliegt hängt von dem Vergleich der acht Eckpunkte mit dem Schwellenwert ab.

Da die MHD-Daten, die zur Verfügung standen, sehr grob gerastert waren (vergleiche Einführung), wurde linear interpoliert, um die Werte an den Eckpunkten zu bestimmen. Die Interpolation wird noch an weiteren Stellen im Visualisierungsprogramm verwendet. So sind auch die Farbverläufe bei der Darstellung skalarer Daten, z.B. der Temperatur, der Interpolation zu verdanken. Der Marching-Cube Algo-

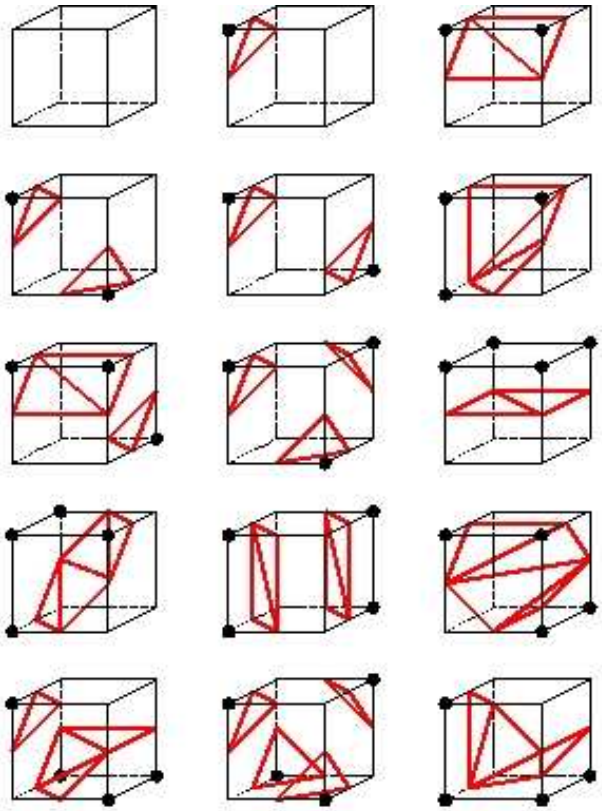


Abbildung 2: Die fünfzehn Fälle von Lorenson und Cline

rithmus kann im Prinzip beliebig genau die Oberfläche finden und darstellen. Die Zeit für die Berechnung der Isoflächen hängt stark von der gewünschten Genauigkeit und damit von der Rate, mit welcher der Volumendatensatz im Marching-Cube Algorithmus abgetastet wird, ab. Es wurden Untersuchungen durchgeführt, bei welcher Auflösung und der damit verbundenen Schleifen-Iterationszahl im Programm ein guter Mittelwert zwischen Performance und Optik liegt. Abbildung 3 zeigt, wie viele Dreiecke pro Anzahl der Schleifendurchläufe gefunden wurden. Die Schleifendurchläufe sind hierbei direkt kubisch proportional zur Dimensionsauflösung. Dabei bedeutet eine Dimensionsauflösung von z. B. 10, dass jede Raumrichtung mit zehn Teilschritten durchwandert wird. Das Laufzeitverhalten wird immer schlechter, je höher man die Auflösung wählt. Die Laufzeit wächst praktisch linear mit der Anzahl der Schleifendurchläufe an. Ab einer Auflösung von 20 wurde das Programm merklich langsamer.

Damit das im Marching-Cube Algorithmus erzeugte Dreiecksnetz in OpenGL unter Berücksichtigung von Beleuchtung gezeichnet werden kann, müssen die Oberflächennormalen des Netzes an den Eckpunkten der Dreiecke vorliegen. Dies ist nötig, da in die Berechnung der Beleuchtungseffekte der Einfallswinkel von Lichtstrahlen auf der beleuchteten Oberfläche einfließt. OpenGL funktioniert prozedural, d.h. es benutzt einen Mechanismus, den man sich als ’processing pipeline’ vorstellen kann (siehe [3]). Schickt man einmal eine Farbe in die Pipe, zum Beispiel Rot mit:

```
glColor3f(1.0, 0.0, 0.0);
```

so werden alle weiteren Punkte, Linien und Polygone in rot gezeichnet, bis dies durch ein erneutes Setzen der Farbe wieder geändert wird. Genau so ist es auch mit dem Befehl, der die Normale setzt:

```
glNormal3f(x, y, z);
```

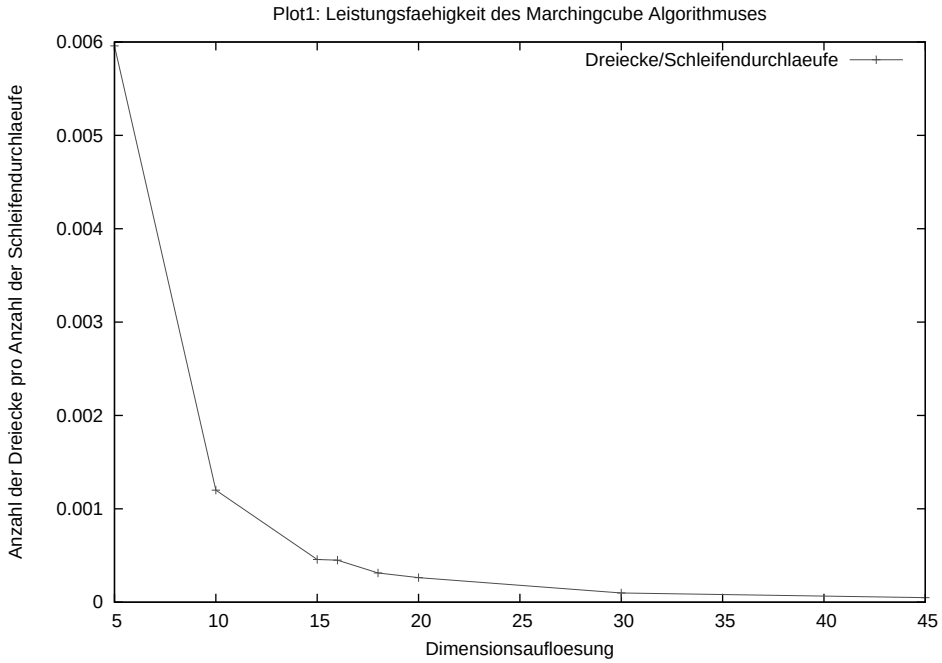


Abbildung 3: Effizienz des Marching-Cube Algorithmuses in Abhängigkeit der Dimensionsauflösung

Bei den Oberflächen, die bisher gezeichnet wurden, lag bzgl. der Oberflächennormalen ein Spezialfall vor. Es handelte sich um die Oberfläche der Einheitskugel (Globus), für die die Oberflächennormalen gerade gleich den Punktkoordinaten sind. Daher war es bisher ausreichend, die Normale in Richtung des zu zeichnenden Punktes zu setzen:

```
glNormal3f(x, y, z);
glVertex3f(x, y, z);
```

Dies erzeugt eine korrekt beleuchtete Kugeloberfläche, da der Ursprung des OpenGL-Koordinatensystems im Mittelpunkt der Kugel liegt. Für die Isoflächen ist dies aber nicht korrekt, da diese Flächen eine beliebige Ausrichtung im Raum haben. Dies verdeutlicht Abbildung 4. Diese Bildserie zeigt die Isofläche zuerst mit Punkten als Normalen, dann mit den Normalen der tatsächlichen Marching-Cube Dreiecke und schließlich mit gemittelten Normalen unter Berücksichtigung der Nachbar-Dreiecke. Da alle Darstellungsarten ihre Vor- und Nachteile haben, kann man im Programm diese Anzeige-Art verändern. Der Algorithmus zum Mitteln der Normalen ist zum Beispiel langsamer als die anderen beiden, erzeugt aber optisch eine glatte Oberfläche. Wählt man die direkten Dreiecks-Normalen als Eckpunktnormalen, so ergibt sich zwar eine schnelle Berechnung, die Oberfläche sieht aber nicht mehr glatt aus.

Da der Marching-Cube Algorithmus die Dreiecke, beziehungsweise die Eckpunkte der Dreiecke nicht sortiert ausgibt, musste man, um die Normalen zu glätten, erst einmal gleiche Eckpunkte finden. Erst dann kann man die Normalen miteinander verrechnen. In einem ersten Ansatz brauchte die Suche nach den gleichen Eckpunkten quadratische Laufzeit, da jeder Eckpunkt mit jedem verglichen wurde. Dies konnte jedoch drastisch optimiert werden dadurch, dass erstens die innere Schleife nicht mehr von Null bis zum Ende geht, sondern nur noch vom aktuellen Wert der äußeren Schleife bis zum Ende und zweitens nur noch dort nach passenden Eckpunkten gesucht wurde, wo der Marching-Cube Algorithmus auch welche gefunden hat. Abbildung 5 zeigt, wie drastisch dies die Anzahl von Schleifendurchläufen reduziert hat.

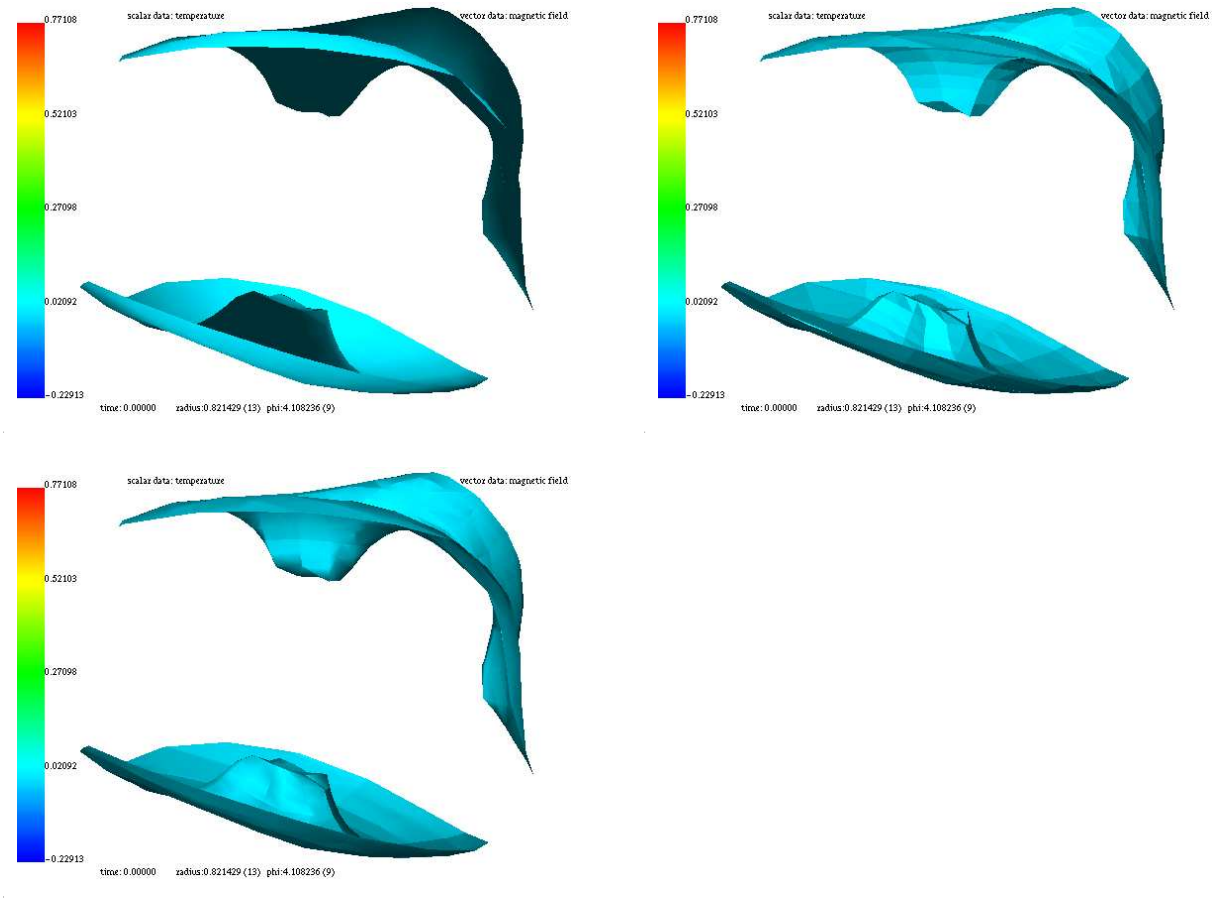


Abbildung 4: Bilder-Serie: Das erste Bild zeigt die Isofläche, bei der die Normalen gleich den Punktkoordinaten gesetzt wurden, rechts daneben wurde für jedes im Marching-Cube Algorithmus gefundene Dreieck die Normale berechnet und im letzten Bild wurde diese dann noch mit den Nachbarnormalen gemittelt.

Trackingsystem

Um eine virtuelle Welt zu schaffen, ist es sehr wichtig, dass der Benutzer sich in diese Welt integriert fühlt. Um dies zu ermöglichen muss die Software die Bilder, die sie ausgibt, den Bewegungen und Tätigkeiten des Benutzers anpassen. Dies bedeutet zum Beispiel, dass ein Objekt größer wird, wenn man auf es zu geht. Ein weiterer wichtiger Teil ist die Kontrolle über die Gegenstände der Virtual-Reality, so dass hier Bewegungen nicht nur zweidimensional wie mit einer herkömmlichen Maus, sondern dreidimensional interpretiert werden müssen. Beide Aufgaben werden durch so genannte Trackingsysteme gelöst, welche die Position und die Ausrichtung von am Trackingsystem angeschlossenen Sensoren messen können. Hier kam das Trackingsystem 3Space$^{(R)}$ Fastrak$^{(R)}$ von Polhemus zum Einsatz. Dieses besteht aus mehreren Komponenten, die im folgenden kurz vorgestellt werden, um die Funktionsweise des Systems zu verdeutlichen. Der Artikel beschränkt sich dabei auf die englischen Original-Namen.

- Systems Electronic Unit (SEU)
- Transmitter
- mehrere Receiver (Sensoren), z.B. der 3D-Eingabestift Stylus sowie der Sensor für das Kopftracking

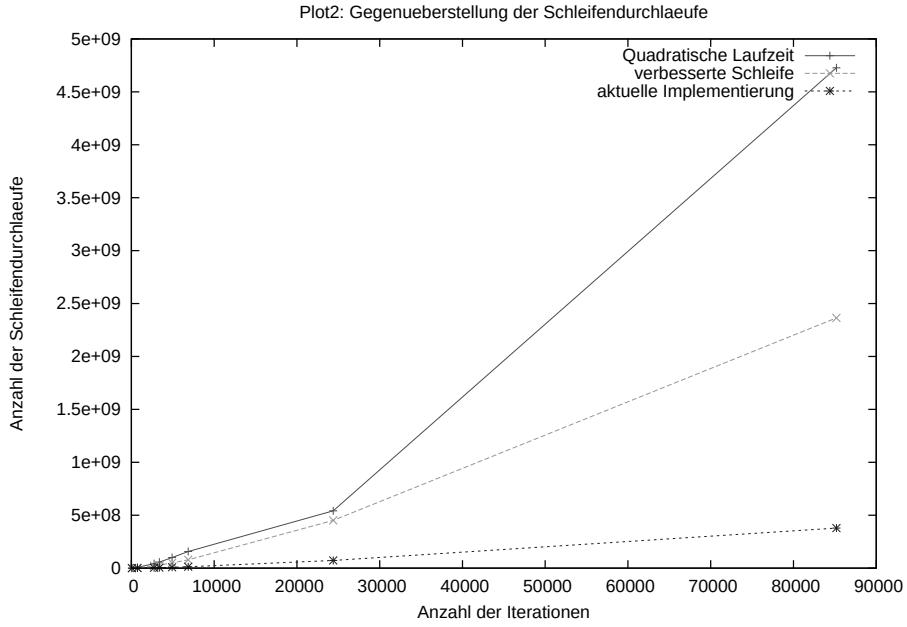


Abbildung 5: Laufzeit-Verbesserung der Normalen-Glättung des Marching-Cube Algorithmus

Die SEU ist das Herzstück des Systems; sie führt die zur Positionsbestimmung nötigen Berechnungen durch und gibt diese an den PC weiter. Im hier vorliegenden Aufbau wird dazu die serielle Schnittstelle (RS-232) verwendet. An die SEU wird der so genannte Transmitter angeschlossen. Dieser generiert niedrigfrequente Nahfeld-Magnetfelder und zwar mittels drei stationärer Antennen. Dieses Feld wird von den Receivern gemessen, abermals mit drei Antennen. Diese übermitteln ihre Daten zurück an die SEU, welche die Daten dann verarbeitet. Das Gerät arbeitet auf einer Trägerfrequenz von 12019 Hz und schafft 120 updates/Sekunde/Receiver.

Beim Anschluss des Gerätes an ein Linux-System muß beachtet werden, dass die Rechte für den seriellen Port (oft auch als COM-Port bezeichnet) für den Benutzer nicht eingeschränkt sind. Dies kann über folgenden Befehl, der vom Super-User ausgeführt werden muss, erreicht werden:

```
chmod +rw,o /dev/ttyS0
```

Dabei ist /dev/ttyS0 der erste serielle Port. Der Befehl bewirkt, dass sämtliche Nutzer auf den Port lesen und schreiben dürfen. Es hat sich später heraus gestellt, dass dies eigentlich nur für einen Test der Kommunikation nötig zu sein scheint. Um den Tracker zu testen, kann man einfach ein ’P’ an das Gerät schicken. ’P’ ist hierbei der Befehl, der das Trackingsystem anweist, die aktuellen Sensorparameter über die serielle Schnittstelle zu schicken. Als Antwort erhält man die Koordinaten der Receiver. Zum Beispiel:

```
cat /dev/ttyS0 &
echo "P" > /dev/ttyS0
01   10.40   0.97  24.56 155.05 -65.23-127.12
02    9.88   1.71  -3.20 -12.90  54.99-115.75
03    2.90  -3.14  20.31-165.65  -1.36  80.34
```

Der Ausgabe kann entnommen werden, dass drei Receiver angeschlossen sind. Die Einheiten der Translationswerte sind standardmäßig Inches, dies lässt sich aber auf Zentimeter ändern. Es gibt auch die

Möglichkeit, ungewollte Werte auszublenden, bzw. Receiver zu deaktivieren. Im diesem Beispiel hat jeder Receiver seine x-, y- und z-Koordinate gesendet so wie den Azimuth-, Neigungs- und Rollwinkel jeweils in Grad.

Bei der Interpretation der vom Polhemus-Fastrak Tracker gelieferten Werte muss beachtet werden, dass das System die Position nur eindeutig in einer Halbkugel um den Transmitter herum bestimmen kann. Beim Verlassen und wieder Eintreten in diese Hemisphäre kommt es zu Wert-Sprüngen. Daher muss die Hemisphäre so eingestellt werden, dass der Benutzer sie während der Arbeit mit dem Tracking-System typischerweise nicht verläßt. Auf den Transmitter ist ein Koordinatensystem gedruckt, welches dabei hilft, die Hemisphäre zu wählen. Mit dem Befehl:

```
echo "H3, 1.0, 0.0, 0.0" > /dev/ttyS0
```

setzt man die Hemisphäre für den dritten Receiver in Richtung der X-Achse, so dass es keine Sprünge gibt, so lange man im Positiven bzw. Negativen X bleibt. Es gibt auch noch die Einschränkung, dass nur an Receiver-Port 1 der Button des 3D-Eingabestiftes Stylus funktioniert. Weiteres ist [6] zu entnehmen.

OpenGL Feinheiten

Wie schon im Abschnitt über die Visualisierung von Isoflächen angedeutet, ist OpenGL keine beschreibende, sondern eine prozedurale Architektur. Dies hat zur Folge, dass bei der Umsetzung von Translationen, Rotationen, usw. genau auf die Reihenfolge geachtet werden muss. Ausser 3D-Transformationen geht in die Darstellung einer Szene auch eine Projektion ein. In diesem Abschnitt soll geklärt werden, welche Art der Perspektive für ein Virtual-Reality System zu wählen ist und wie dann das Bild zu Stande kommt.

In OpenGL gibt es drei Darstellungsmodi, die festlegen, wie eine berechnete Szene auf dem Bildschirm angezeigt wird. Diese sind:

```
gluPerspective(), glFrustum(), glOrtho().
```

$gluPerspective()$ und $glFrustum()$ entsprechen unserer normalen Wahrnehmung und erzeugen eine perspektivische Projektion, d.h. Objekte, die weiter hinten sind, werden kleiner gezeichnet. $glOrtho()$ hingegen realisiert eine Orthogonal-Projektion, die die Welt ohne Flucht-Punkt zeigt. Diese wird zum Beispiel in der Architektur verwendet, da sie Größenverhältnisse nicht zerstört. Im Gegensatz zu $gluPerspective()$ kann man mit $glFrustum()$ sein Blickfeld frei bestimmen. Während $gluPerspective()$ immer frontal auf die Szene schaut und von einer symmetrischen Sichtpyramide ausgeht, kann mit $glFrustum()$ eine beliebige Sichtpyramide definiert werden, in welche die aktuelle Tracker-Position einfließt. Ein einfaches Beispiel kann in [4] auf Seite 124 gefunden werden. Der aktuelle Aufruf, berechnet mit dem Strahlen-Satz

$$\frac{MYNEAR}{trak\_pos\_z} = \frac{Kantelinks}{screenborderleft - trak\_pos\_x}, \tag{1}$$

sieht dann so aus:

```
ratio = MYNEAR/trak_pos_z;       //nach Strahlen-Satz
glFrustum(ratio*(screenborderleft - trak_pos_x),   //Kante links
          ratio*(screenborderright - trak_pos_x),  //Kante rechts
          ratio*(screenborderbottom - trak_pos_y), //Kante unten
          ratio*(screenbordertop -trak_pos_y),     //Kante oben
          MYNEAR, MYFAR);                          //cliping-planes
```

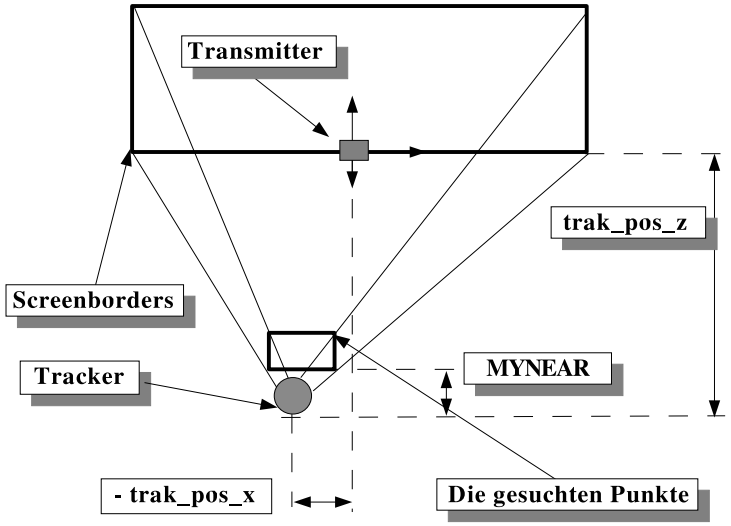


Abbildung 6: Skizze zur Erklärung des glFrustum()-Befehls

Damit hat man das Sichtfeld, aber noch ist das Bild nicht stereoskopisch. Um das Bild stereoskopisch darzustellen, standen schon einige Möglichkeiten zur Verfügung, allerdings wurde das Trackingsystem noch nicht berücksichtigt. Die verschiedenen Anzeigemodi sind im Programm wie folgt definiert:

```
typedef enum { MONO, STEREO_ACTIVE, STEREO_LEFTRIGHT } tStereoMode;
```

Während $STEREO\_LEFTRIGHT$ für ein überbreites Fenster auf zwei Monitoren gedacht ist und auf beiden das Bild zeichnet, funktioniert $STEREO\_ACTIVE$ durch das Verwenden von zwei Framebuffern. Der Framebuffer enthält in OpenGL die fertige Szene, ist also das Ende der processing pipeline und enthält die Daten, die an den Monitor geschickt werden. Besitzt man eine Grafikkarte, die einen so genannten Quad-Buffer Stereo-Modus unterstützt, so stehen getrennte Framebuffer für das linke und das rechte Auge zur Verfügung. Das Ansprechen dieser zwei Framebuffer geschieht schematisch so:

```
...
glTranslatef(linkes_Auge);  //Setze die Kamera für das linke Auge
glDrawBuffer(GL_BACK_LEFT); //In den linken Framebuffer schreiben
drawScene();
glTranslatef(rechtes_Auge);  //Setzt die Kamera für das rechte Auge
glDrawBuffer(GL_BACK_RIGHT); //Ab hier in den Rechten
drawScene();
...
```

Um die Kamera unter Berücksichtigung der Positionen der zwei Augen des Betrachters richtig zu setzten, wurde wieder auf die Tracker-Daten zurückgegriffen. Es wurde diesmal aber nicht nur die Position berücksichtigt, sondern auch die Rotation (Neigung) des Sensors. Diese Rotationen liefert der Tracker in Form von Quaternionen. Eine Quaternion kann in die entsprechende Rotationsmatrix umgerechnet werden ([6], Seite 163). Diese Rotationsmatrix wird mit dem Vektor, welcher die Position des linken bzw. rechten Auges relativ zum Trackingsensor angibt, multipliziert:

$$\begin{pmatrix} eye\_x \\ eye\_y \\ eye\_z \end{pmatrix} = \begin{pmatrix} q_0^2+q_1^2-q_2^2-q_3^2 & 2(q_1q_2-q_0q_3) & 2(q_1q_3+q_0q_2) \\ 2(q_0q_3+q_1q_2) & q_0^2-q_1^2+q_2^2-q_3^2 & 2(q_2q_3-q_0q_1) \\ 2(q_1q_3-q_0q_2) & 2(q_0q_1+q_2q_3) & q_0^2-q_1^2-q_2^2+q_3^2 \end{pmatrix} \times \begin{pmatrix} eyeshiftx \\ eyeshifty \\ eyeshiftz \end{pmatrix}$$

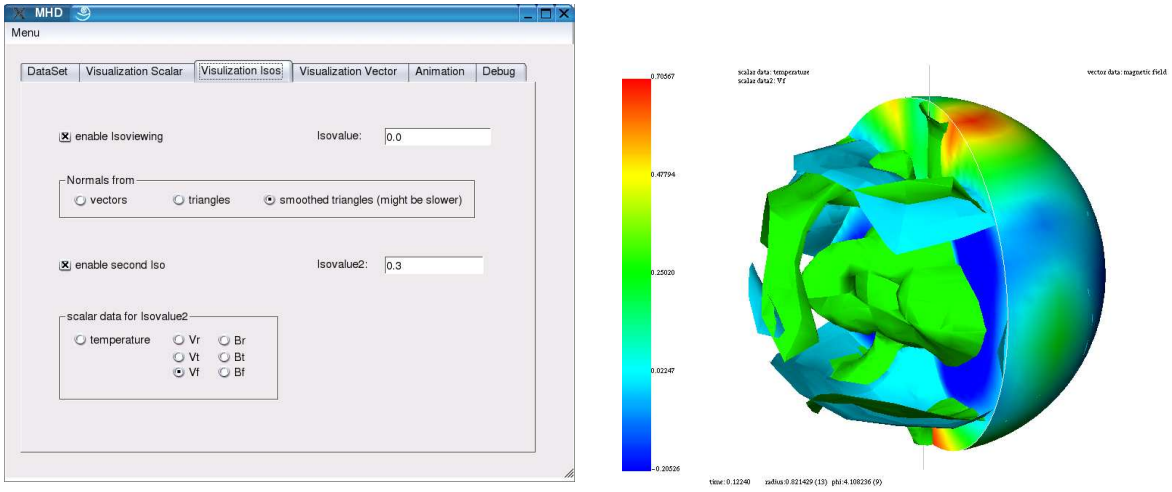


Abbildung 7: Links die grafische Benutzer-Schnittstelle mit den neuen Auswahlmöglichkeiten. Auf der rechten Seite das dazu entsprechende Bild, mit eingeschaltetem ’sphere slicing’.

Danach muss man nur noch

```
glTranslatef(eye_x, eye_y, eye_z);
```

ausführen und für die $eyeshifts$ einmal die Werte fürs linke und einmal fürs rechte Auge einsetzen.

Ausblick

Als erstes ist anzumerken, dass ein Visualisierungsprogramm wie das hier beschriebene wohl immer „work in progress“ sein wird, und dass es immer etwas gibt, was noch fehlt. Im Folgenden wird kurz eine Liste von Funktionen genannt, die es beispielsweise noch zu implementieren gibt:

- Anzeige mehrerer Isoflächen von unterschiedlichen Feldern
- Anzeige mehrerer Vektor-Daten
- Vereinheitlichtes Daten-Einlesen, so dass in Zukunft auch andere Daten berücksichtigt werden können
- Beschleunigung der Software im Allgemeinen und höhere Stabilität im Zusammenhang mit dem Trackingsystem

Das Visualisieren von zwei Isoflächen wurde im Rahmen dieser Arbeit bereits implementiert, jedoch mit der Einschränkung, dass dies bei eingeschalteter Normalen-Glättung nicht richtig funktioniert. Die zwei Flächen sind in Abbildung 7 dokumentiert, so wie die von mir neu implementierten Wahlmöglichkeiten.

Danksagung

Bedanken möchte ich mich bei allen, die mich bei meiner Arbeit hier am Forschungszentrum tatkräftig unterstützt haben. Dazu gehören neben meinem Betreuer Dr. Herwig Zilken, dem der meiste Dank gebührt, auch mein Büro-Nachbar Dr. Oliver Passon, die Dispatchmitarbeiter und Dr. Rüdiger Esser, der stets ein offenes Ohr hatte für die Probleme der Gast-Studenten. Dank gilt auch Dr. Maxim Reshetnyak, der neben der Idee auch die Daten zu diesem Projekt geliefert hat.

Literatur

1. M. Reshetnyak, B. Steffen, The subgrid problem of thermal convection in the earth's liquid core, Numerical Methods and Programming (2004) Vol. 5.
2. W. E. Lorenson, H. E. Cline, Marching Cubes: A high resolution 3D surface construction algorithm, ACM Computer Graphics (1987), Volume 21 Nr. 4.
3. D. Shreiner, OpenGL Reference Manual, Third Edition, Addison-Wesley (1999).
4. Woo, Neider, Davis, Shreiner, OpenGL Programming Guide, Third Edition, Addison-Wesley (1999).
5. F. Roth, Algorithmen zur Konstruktion von Isoflächen aus dreidimensionalen Volumendatensätzen, Diplomarbeit im Fach Informatik an der TH Aachen (1999).
6. 3Space$^{(R)}$ User's Manual, Polhemus (1993).

A Parallel Cluster Algorithm for Monte Carlo Simulations Applied to Model DNA Systems

Jakob Schluttig

University of Leipzig
Institute for Theoretical Physics

eMail: jakob.schluttig@itp.uni-leipzig.de

Abstract:

Cluster methods are used in Monte Carlo simulations to decrease the autocorrelation time, i.e. the interval between statistically independent configurations, which becomes crucial close to critical points and phase transitions. The aim of this work is to build the basis for a Monte Carlo cluster algorithm for continuous two dimensional spin systems. First the Kornyshev-Leikin model potential is introduced which is applied to Monte Carlo simulations of DNA systems. Afterwards a purely geometrical technique for searching clusters is described. Furthermore it is extended to an energetic cluster criterion, which is the basis in Monte Carlo cluster methods. The scaling of the implementation is measured and analyzed. Finally it is used to study geometric clusters as a function of different DNA characteristics, e.g. the charge compensation parameter θ.

Kornyshev-Leikin Pair Potential for Rigid Helical Molecules

It is well known that DNA forms close packed aggregates of various structures, e.g. in human chromosomes or viruses. Experimentally it was observed that short fragments form columnar aggregates which are suitable to study interactions, e.g. like charge attractions between molecules and global structures.

At first glance a whole DNA is far too complex to describe its interaction with other molecules in a closed analytical framework. However, A. A. Kornyshev and S. Leikin [1] described DNA molecules as long cylinders, carrying helical, continuous line charges on their surface, taking advantage of the symmetries in helical molecules. Thus it was possible to derive an exact formalism which can be used to calculate interactions between two stranded helical molecules like the DNA. The theory in [1] is formulated in a rather general way, so the potential that is finally used for the simulation had to be derived and adapted to the actual application. The whole interaction energy is obtained by a sum of three different terms. The first one, labeled as w_{cyl}, corresponds to the interaction between two homogeneously charged cylinders. w_{self} is a "self correlation" energy, which is due to correlated discrete surface charge distributions on each molecule. Ultimately w_{cross} is a "cross correlation" energy, which is caused by nonrandom alignment of discrete charges on the opposing molecules. The following formulae describe an energy density, where the energy is normalized to the persistence length L_p [2].

$$u(\phi, r) = \frac{(4\pi\sigma_0)^2}{\epsilon}\left[w_{\text{cyl}}(r) + w_{\text{self}}(r) + w_{\text{cross}}(r,\phi)\right] \tag{1}$$

$$w_{\text{cyl}}(r) = \frac{(1-\theta)^2}{2\kappa^2}\frac{K_0(\kappa r)}{[K_1(\kappa a)]^2} \tag{2}$$

$$w_{\text{self}}(r) = 4\sum_{n=1}^{\infty}\cos^2(n\tilde{\phi}_s)\frac{\sum_{j=-\infty}^{\infty}\frac{I_{j-1}(\kappa_n a)+I_{j+1}(\kappa_n a)}{K_{j-1}(\kappa_n a)+K_{j+1}(\kappa_n a)}K_{n-j}^2(\kappa_n r)}{\left[K_{n-1}(\kappa_n a)+K_{n+1}(\kappa_n a)\right]^2\kappa_n^2} \tag{3}$$

$$w_{\text{cross}}(r,\phi) = 4\sum_{n=1}^{\infty}(-1)^n\cos(n\phi)\cos^2(n\tilde{\phi}_s)\frac{K_0(\kappa_n r)}{\left[K_{n-1}(\kappa_n a)+K_{n+1}(\kappa_n a)\right]^2\kappa_n^2} \tag{4}$$

$$\kappa_n = \sqrt{\kappa^2 + \left(\frac{2\pi n}{H}\right)^2} \tag{5}$$

Due to the very rapid convergence, the sums in eqs. 3 and 4 may be truncated at $n = 5$ and $-5 \leq j \leq 5$.

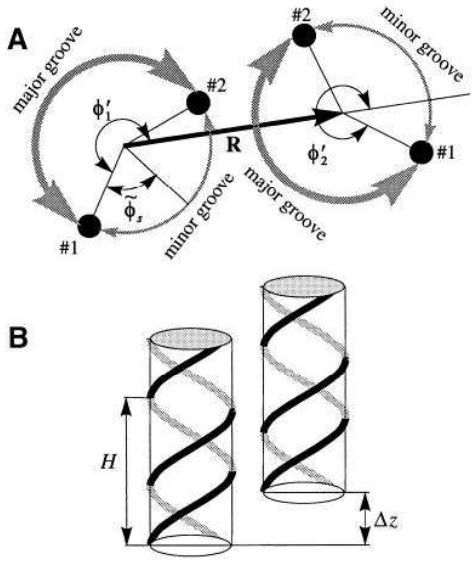


Figure 1: Simple scheme of important structural values for two interacting DNA double strands.

The parameters appearing in eqs. 1-5 are briefly explained [5]:
First there are the structural parameters of the phosphate pattern (see fig. 1, taken from [4]): the helical pitch H, the azimuthal half-width of the minor groove $\tilde{\phi}_s$ and the hard-core radius a. Furthermore each DNA duplex carries the negative charge of phosphates with surface charge density σ_0 plus a compensating positive charge arising from adsorbed counter ions. The degree of compensation is described by the parameter θ, where $0 \leq \theta \leq 1$. In eqn. 2 the term w_{cyl} vanishes if $\theta = 1$. The mobile counter ions in solution cause an exponential decay of the Coulomb interaction of the two helices for large separations. This exponential decay is parameterized by the inverse Debye screening length κ. The solution is also considered by its dielectric constant ϵ. Actually the dielectric constant and the Debye screening length are both temperature dependent and κ is also a function of ϵ. Although this is not taken into account here [3].

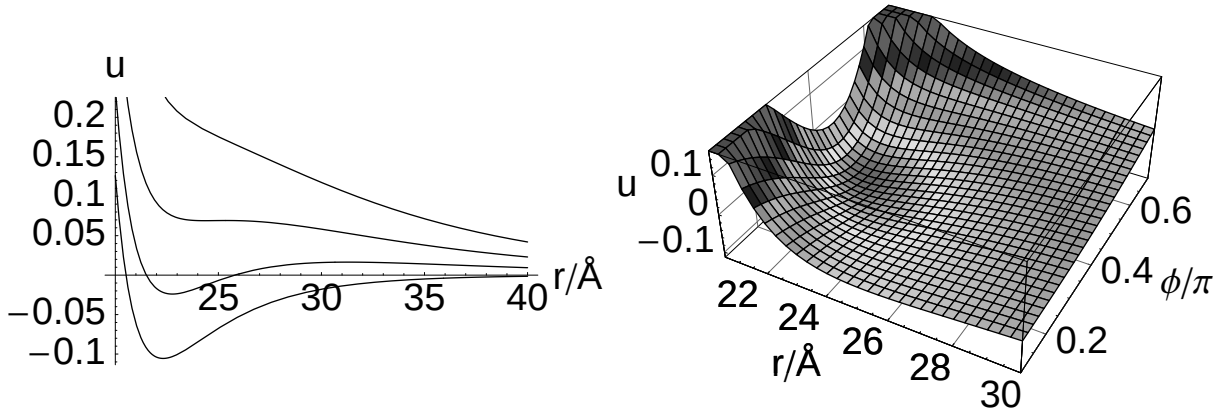
The simulations were carried out considering B-DNA structure. The proper parameters were taken from [4] and are collected in table 1.

L [Å]	a [Å]	H [Å]	$\tilde{\phi}_s/\pi$ [rad]	σ_0 [μ C/cm^2]	ϵ
500.0	9.0	33.8	0.4	16.8	80

Table 1: Structural and chemical parameters for the DNA-B molecules.

From eqs. 2 to 4 it can be seen that there is a two dimensional potential energy landscape for a pair of DNA molecules depending on the distance of two strands $r = |\mathbf{R}|$ and the relative azimuthal orientation ϕ. The latter can be simply calculated as the difference of the respective angles ϕ_1' and ϕ_2' of the $5' \rightarrow 3'$ strand, relative to a reference direction. A shift in the axial direction Δz therefore translates into a different azimuthal orientation.

Due to a lack of time it was not possible to study the DNA aggregates with respect to different screening lengths κ^{-1}. For all discussions and measurements in this report it is fixed to $\kappa = 0.1\text{Å}^{-1}$. The following pictures should help to get an imagination of the potential energy and support the understanding of the expected effects.



a **b**

Figure 2: **a**: Kornyshev-Leikin potential at $\phi = 0.4\pi$ for $\theta = \{1.0, 0.9, 0.85, 0.8\}$, where $\theta = 1.0$ is the lowest plot and $\theta = 0.8$ the uppermost. **b**: Calculated with $\theta = 1.0$. The values of the potential energy are expressed in artificial units.

Fig. 2.**a** shows four qualitative different types of the interaction potential obtained for different values of θ. For high compensation θ the interaction of the two DNA strands is purely attractive for a nearly perpendicular azimuthal alignment until a certain equilibrium distance of about 22Å is reached. That is very short considering the hard core distance of $2 \times 9\text{Å} = 18\text{Å}$. As the number of adsorbed counter ions decreases a local maximum arises which separates a condensed and a crystalline state, while the local minimum is still below zero and the energy barrier gets lower. For even lower values of θ the local minimum at short distances exceeds zero and thus the bound (clustered) state is obviously not any longer more preferable than the crystal state. Finally there is a certain - probably critical - point of charge compensation where the local minimum vanishes and the potential gets completely repulsive.

Keeping in mind that the potential energy landscape is not 1- but 2- dimensional (see fig. 2.**b**) it becomes clear that even in the case of $\theta = 1$ it is not sure whether all the DNA strands will go into the equilibrium distance and form one big cluster in the ground state. The preferred orientation between two molecules at short distances is $\approx \pi/2$ (cmp. fig. 2.**b**). A clustering of three molecules will result in an energetically

frustrated configuration and therefore the formation of big clusters is not obvious. Also it can be seen from fig. 2.**b** that for large distances of two interacting strands the parallel orientation is favorable. Interestingly even if the system would be simulated on a lattice the potential would therefore still be density dependent. This makes it clear that there is a rich phase behavior to be expected. For $\theta = 0.7$, i.e. in the repulsive regime, this was studied for the ground state in ref. [5].

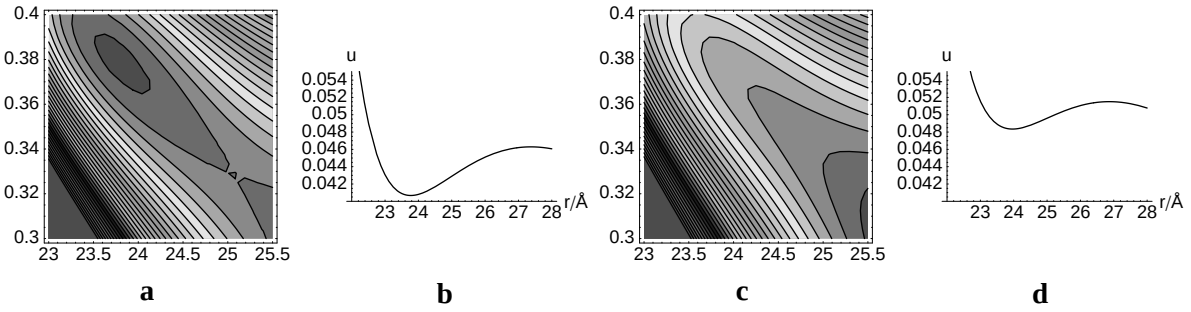


Figure 3: These pictures illustrate the critical behavior of the Kornyshev-Leikin potential at $\kappa = 0.1$ and $\theta \approx 0.86$. Also it is shown that it is important to look for a local minimum with respect to r *and* ϕ. **a**, **b**: $\theta = 0.864$. **c**, **d**: $\theta = 0.860$. **b** and **d** are both obtained for $\phi = 0.38$.

Another important fact that results from the dependency of the orientation ϕ is visualized in fig. 3. The two plots **b** and **d** show the potential for fixed mutual orientations. They seem to be qualitatively similar. However, this is not so, which is proven by **a** and **c** from where it is obvious that for $\theta = 0.860$ there *is no* potential barrier as seen in fig. 3.**d** since the two strands can always loose energy by orientating parallel and separate from each other at the same time. Whereas for $\theta = 0.864$ a real local minimum exists. This demonstrates that the critical value for θ must be in the range of $\theta_c \in [0.86, 0.864]$ and not at 0.85 as it could be expected from fig. 2.**a**.

Simulations

All simulations were carried out with the program *"SpinCG^{2d}"* by G. Sutmann [6]. As it is seen from eqs. 1-4, the only variables describing the interaction between two DNA molecules are their interaxial distance and their mutual orientation. This picture corresponds to a 2-dimensional spin system, where spins have three degrees of freedom (position, orientation), i.e. a kind of generalized X-Y-Model.

The starting configuration of DNA strands consists of a hexagonal structure with lattice constant d. This distance is related to the DNA density. Afterwards a mixture of down-hill and simulated annealing algorithm is performed, i.e. the temperature is decreased by a certain amount after *each* Monte Carlo step. In so doing the system is cooled down from $T(t_{\mathrm{MC}} = 1) = 1000$K to $T(t_{\mathrm{MC}} = N_{\mathrm{MC}}) = 30$K. N_{MC} is the number of Monte Carlo steps performed in a whole simulation and t_{MC} is the Monte Carlo time. This means that not only the system is given no time to equilibrate but in the whole simulation the forming of an equilibrium state is actually prevented. On the other hand as we are mostly interested in energetically favorable states the cooling to very low temperatures will definitely lead to ground state like structures of the aggregate. These are naturally somehow artificial since the DNA would probably change its configuration dramatically at such low temperatures. It can be assumed that for long simulation times N_{MC} the influence of the non equilibrating kind of the simulation can be neglected. In [2] the temperature was kept constant over a certain number of Monte Carlo steps and similar results were observed. During the simulation the size of the trial moves is adapted to have an acceptance rate of 0.5. Since the explicit evaluation of the potential is rather expensive it is interpolated during the simulation by a second order interpolation from a table. For the cross correlation energy the first 5 terms of the sum

over n are interpolated and multiplied by the orientation dependent $\cos(n\phi)$.

Geometrical Cluster Search

Thinking of an efficient way to identify neighbors in a certain distance, which still has to be defined, it is most important to consider the continuous properties of the particles in the system on the one hand and the spatial decomposition of *"SpinCG^{2d}"* on the other. The latter is mostly significant for the parallelization. The whole program is written using MPI for distributed memory systems without memory replication, so the cluster search also has to deal with this distributed data handling.

The basic approach consists of:

1. identifying clusters sequentially and independently on each processor
2. communicate with other processors to link global clusters
3. scatter the whole linking information

First of all it is necessary to introduce a geometric cluster criterion. If the distance of two molecules exceeds a threshold length r_0 they are not considered as *neighbors*. Since the structures of aggregate configurations are extremely varying, it seems to be a good choice to correlate this criterion with the Kornyshev-Leikin potential and thus introduce a barrier dependent threshold length r_0. By some test runs it turned out that postulating r_0 as the distance where the potential barrier is overcome by $4/5$ provided acceptable results (see fig. 4). Because of the form of the potential it was always quite obvious which particles were in the short distance of the potential minimum, and which were in a kind of crystalline state with respect to each other. That is the clusters were clearly separated and the identification could be verified easily.

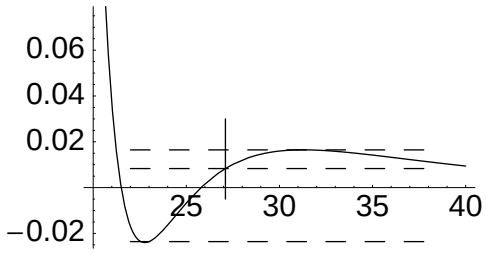


Figure 4: The maximum distance of two particles belonging to one cluster is derived from the interaction potential.

Sequential Local Cluster Identification

There is a variety of algorithms for searching clusters on a lattice. One of the most well known is probably the Hoshen-Kopelman algorithm [7]. A lattice system is particularly simple in that way, that firstly the particles (i.e. the lattice sites) are in a given order which makes them easy to address. Furthermore there is a definite number of neighbors at given positions. Both of these lattice properties are not given in the considered system. Since there is no way to order the particles systematically, it would be necessary to check every particle against every other particle which would result in a complexity of $\mathcal{O}(N^2)$. Even if the particles would be sorted by the x- or y-coordinate, the quadratic scaling behavior would probably be

only decreased by a factor. This is of course undesirable. Thus the idea is to alter a lattice based algorithm to fit the needs without loosing its benefits.

Virtual Lattice Structure

This is accomplished by introducing a virtual lattice and sort the molecules into the lattice cells by a linked list (see fig. 5). In so doing a small number of particles belonging to one cell can be addressed nearly as fast as if they actually were particles in a lattice system. Furthermore this method makes it possible to check only pairs of molecules in a number of potential neighbor cells. But the structure of the lattice still needs some more investigation.

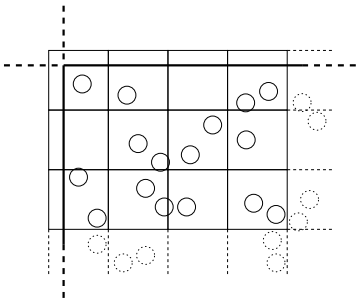


Figure 5: Schematic view of the sorting of particles into a virtual lattice at the border of a processor.

It is plausible that the preferable structure is a square lattice. It is easy to address and the particles can be assigned efficiently to the cells. How to choose the lattice constant l? A very small l means that in one lattice cell there will be only few molecules and hence the step of checking each particle in one cell against any particle in another one will not be expensive although it has a complexity of $\mathcal{O}(N_c^2)$, where N_c is the number of particles in a cell. On the other hand for large l the number of cells in the whole lattice *and* also the number of possible neighbor cells decreases. The latter two factors are both scaling with $1/l^2$. Whereas the number of particles N_c is scaling with l^2. So the overall scaling with respect to l will be $1/l^2 \times 1/l^2 \times (l^2)^2 = 1$. From that it is not obvious why the introduction of a lattice should improve the performance.

It has to be noted that every particle has a certain size (hard core radius a in the viewed application) which gives an upper bound for the density of the system, i.e. there is a certain l where on average only one or two particles are located in each cell. Decreasing l even further would result in many empty lattice cells which still have to be checked as potential neighbor cells. On the other hand for very small l there are cells whose particles are always within the radius r_0 which avoids an explicit check of particle pairs within cells. But since for the specific example r_0 is not much larger than twice the hard core radius of one of the molecules it would surely not make sense to decrease l to such low values. So there is a lower bound of l. Also a minimum of 8 neighbor cells exists which always have to be checked because they have bordering corners or edges and could therefore contain molecules bound to molecules from the currently considered cell. So it is also apparent that it is no use in increasing l to very large numbers because the number of neighbor cells does not reduce anymore after reaching 8.

An important fact is that for a lattice constant smaller or equal to $r_0/\sqrt{2}$ it is *sure* that particles belonging to one cell are in the same geometric cluster (see fig. 6). Otherwise this has also to be checked which results in another N_c^2 step for every lattice cell! Taking all these actualities into account it seems to be a good choice to set $l = r_0/\sqrt{2}$. Still it is possible that for certain parameters like extremely low or high densities a change of the virtual lattice constant l could result in some speedup. From fig. 6 it can be seen that for this particular l there are 20 cells possibly containing neighbors. For checking each pair of cells

only once it is enough to check 10 neighbors from each cell. There is no cell that definitely can only contain molecules in the neighboring distance, so every cell has to be checked.

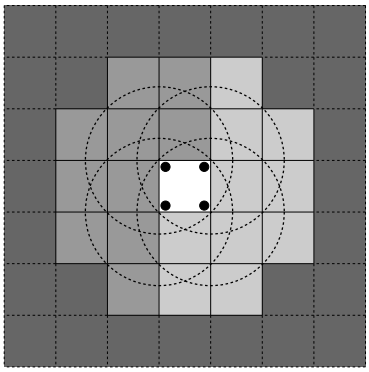


Figure 6: Study of the virtual square lattice with lattice constant $l = r_0/\sqrt{2}$.

For the later communication with other processors it is mandatory that the lattice coordinates are somehow global. Otherwise the linking would get complicated. This is achieved by sorting the particles into a global grid structure, where single grid cells may belong to different processors. Technically this is realized by truncating the decimal places of the two dimensional floating point coordinates of each molecule, which are global, divided by the lattice constant $l = r_0/\sqrt{2}$. In Fortran code it would look like this:

```
lattice_coordinate=FLOOR(real_coordinate/lattice_constant)
```

This also explains the gap between the edge of the processor and the edge of the lattice in fig. 5.

Adapted Hoshen-Kopelman Algorithm

The Hoshen-Kopelman algorithm works iteratively by using a linked list. Fig. 7 is a scheme of the underlying idea. If two clusters are linked, the list entry of the one with the higher *proper* cluster label becomes a pointer to the smaller cluster label, which is represented by a negative integer number. The list entry of the other one contains a positive integer which is the total number of particles in the certain cluster, including all clusters which are linked with this one. Now the *proper* cluster label has to be found in the list by following these pointers until an entry equal or greater than 0 is reached.

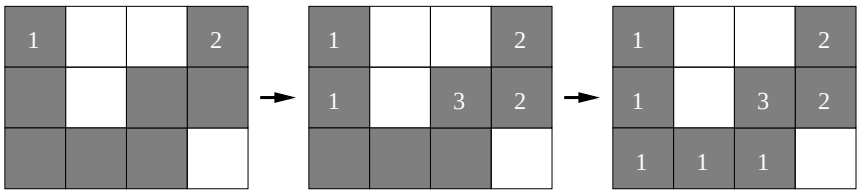


Figure 7: Visualization of the Hoshen-Kopelman cluster search for next neighbors in a small square lattice.

The algorithm is capable of handling any number of potential next neighbor cells. Now some pseudo code will show how the actual sequential local cluster search is done.

```
DO FOR all lattice cells i
  DO FOR half of the possible neighbor cells j
    search for proper cluster label of j
    IF this label is not yet marked as linked with i
      IF there is a pair of molecules from i and j with distance<r_0
        mark label of j as linked with i -> j_l
      END IF
    END IF
  END DO
  IF number of linked clusters is
    0: i gets new cluster label
  >=1: find smallest proper label j_s from {j_l}
       link all other linked neighbors j_l and i to j_s in the list
       save sum of all cluster sizes as new size of j_s
  END IF
END DO
```

For the later parallelization it is important that the newly introduced cluster labels are globally unique. This is no problem since every processor has a non-ambiguous number for identification and there is an upper bound of particles that can be on one processor. So a new cluster label will be calculated like this:

```
cluster_label=local_counter+local_cpu_id*max_particles_per_cpu .
```

To provide the ability of linking local clusters to global ones the linked list array, which will be called `cluster_id` in the following, should have a dimension of

```
number_of_processors*max_particles_per_cpu
```

on every processor. This makes sure that also the linking information is somehow global from the beginning and can therefore be easily exchanged.

Parallel Global Cluster Identification

In *"SpinCG^{2d}"* the whole two dimensional system is divided into N_{PE} domains, where N_{PE} is the number of processors on which the application runs. These domains are not stripes, which would mean that the effort of communication for the Monte Carlo simulation is not decreasing by higher amount of processors, but the program tries to make the domains as close to a square as possible. Hereby through adding more processors the edge length of each processor is reduced which is the driving factor for the communication.

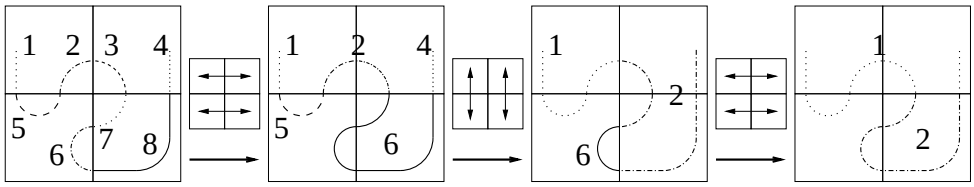


Figure 8: With only next neighbor communication this example would need 5 communication steps until all of the local clusters are connected to one global one.

Now a good strategy has to be found for identifying the global clusters which means clusters that span the domains of more than one processor. Fig. 8 depicts that it is not possible, or at least not efficient,

to do this only by communication between next neighbor processors. The specific example shows that one could probably construct an artificial global cluster to get the necessity of any “desired” number of communication steps until the local clusters would be linked correctly. From that it is obvious that there has to be some kind of *all-to-all* communication.

To realize this with an acceptable scaling the idea is to use a communication tree. While descending to the root no information from the domain borders may be lost. This is achieved by introducing virtual domains as shown in fig. 9. One of two communicating processors is always the *master* which receives the whole border information from the *slave*. Afterwards it processes the bordering edge to link the clusters. This linking is done exactly like the identification of local clusters before. Thereafter it uses the rest of the received information to build up the virtual border of the domain containing the whole area of the two processors before. Since the bordering edge information is already translated into cluster linking pointers and it is obviously not part of the edge of the new virtual domain, it does not have to be sent in the next step of communication and can be rejected. Now the *master* is capable of communicating with other *masters* of the same level in the tree which will have similar dimensions relating to their virtual domains. Thus the number of communication steps is proportional to the logarithm of the number of processors in a certain direction.

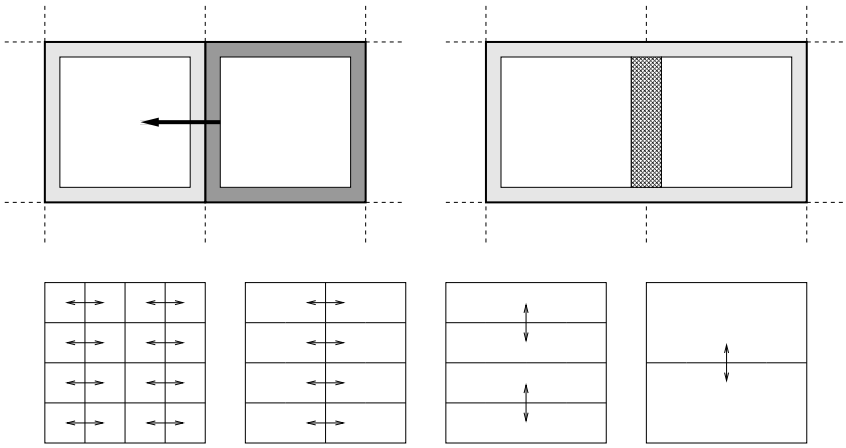


Figure 9: During the communication process the only important thing is the edge of the domain of a processor. These edges grow and become the borders of virtual domains which include the real domains of many processors.

Since communication is very expensive it is important to reduce the information that has to be exchanged to a minimum. First of all there is no way to avoid sending the positions of the particles near the border of the processor domains. The appropriate array will be called `rxyz_border` in the following. Also it is necessary to have the correct cluster labels which are assigned to the bordering lattice cells. And finally the global linking cannot be done without partly knowing the entries of `cluster_id`, i.e. all entries dealing with cluster labels which exist on or which are pointed at from the lattice border. Therefore an array `cluster_id_border` is introduced, whose dimension is two times the number of linked list entries as described before, one for the address and one with the actual value of each entry. In order not to loose any information during the communication, especially concerning the finally broadcast feedback which tells every processor the global labels of the clusters which are contained by it, it is necessary to always keep this information up-to-date with the global linking and *never* reject any of this data, even if it lies on a border inside a virtual domain. This makes up a difference to the other two communication arrays. To reduce the amount of distributed data it is useful to link the clusters lying on the edge of each processor directly to the proper labels before starting the communication. Therefore only proper labels and the labels that they point to, which is important for their size, have to be sent. Fortunately it is always

possible to calculate the dimension of the lattice on the edge of a virtual domain. That is the cluster labels in the lattice can be sent first because there is no uncertainty about the length of the array that has to be received. At the end of the array the size of the other two arrays can be stored, so absolutely no unnecessary information is exchanged.

The amount of communication is dependent on the overall edge length of virtual domains during the whole global cluster linking process. For rectangular areas the optimal form with respect to the edge length would be of course the square. So at first glance it should be faster to build up quadratic virtual domains instead of first linking global lines as it is done in the scheme in fig. 9. But after the global lines are linked, the vertical edges do not contain any relevant information for linking anymore. It is important that for a system with periodic boundary conditions this would be the point of time to apply periodic boundary conditions in horizontal direction. A short calculation example shall confirm the speed up coming from neglecting the vertical edges for further communication. Consider a number of 16 processors and fully quadratic domains with an edge length a. A communication tree which uses quadratic virtual domains if possible would have a whole edge length of $4a + 6a + 8a + 12a = 30a$ to be sent while communicating. Using the approach shown in fig. 9 and rejecting the vertical edge information for the last two communication steps it is only $4a + 6a + 8a + 8a = 26a$. As shown before the size of the linked list table is independent from the chosen type of communication since it is never shrunk.

The hexagonal setup of the system in *"SpinCG^{2d}"* leads to a system that expands more in the y-direction. For that reason it should even be faster to first build up global lines rather than global columns. Since afterwards the amount of communication is only dependent on the length of the lines, which is smaller than the length of columns would be, and does not increase anymore. Also the number of processors in the y-direction is greater than the decomposition in x-direction, that is why the rejection of the vertical borders can be done one communication step earlier under certain circumstances. Finally if we consider the architecture of a parallel computer, e.g. the *ZAMpano* [11], it is likely that neighboring processors in x-direction have shared memory, which makes the first one or two steps of communication fast in the case of first building global lines. But it would need some thorough investigations to prove this.

After all this there will be a *global master*. In its local array `cluster_id` it contains the whole global linking information. If it is enough for every processor to know only the size of clusters whose global labels originate from it, the *global master* must only send back the updated entries of the linked list array which originally came from the specific processor. If in contrast it is important that every processor knows the size of every cluster which can be found in its domain, then the whole linked list array of former edge cluster entries will have to be broadcast to every processor.

Extension to Capability of Building Clusters for a Monte Carlo Cluster Algorithm

An Energetic Cluster Criterion

A very important fact about a Monte Carlo algorithm in general is, that it has to fulfill detailed balance. That is the probability of going from one state into another in a Monte Carlo step must be the same as the probability of getting back. In cluster algorithms this is ensured by the introduction of a certain probability to cut a geometric cluster into smaller parts dependent on energy and temperature as it was postulated by R. H. Swendsen and J. S. Wang in 1986 [8]. Actually every bond between two particles that geometrically belong to one cluster is cut by a probability calculated from their pair energy. The original algorithm dealt with systems of discrete degrees of freedom like the Potts spin models. U. Wolff extended this to a theory which could be applied to continuous spin systems like the $\mathcal{O}(n)$ models in general [9]. To compare two spins his approach uses a projection of the spins to a predefined direction. But still the system has to be simulated on a lattice and the energy is not distance dependent. It is not yet clear, how the criterion of detailed balance can be fulfilled for the cluster building for a system with

positional and angular degrees of freedom exhibiting frustrated configurations.

Necessary Modifications

Due to these considerations the present work was focused to building the basis of a Monte Carlo cluster algorithm. This implies some modifications of the previously described algorithm. The most important difference is that the cluster labels are obviously no longer associated with cells of the virtual lattice but with each molecule in the system since it must be possible that every molecule belongs to an independent cluster, even if two or more of them are assigned to one cell.

A short pseudo code shows the approach for the local cluster identification.

```
initialize every particle with a globally unique cluster label
all clusters have size 1
DO FOR every cell
  DO FOR every pair of molecules in this cell
    IF energetic criterion fulfilled
      get proper labels of both particles
      # ^ ^ important to do this here, can change in every step!
      link referring clusters to smaller proper label, sum size
    END IF
  END DO
  DO FOR every molecule inside this cell
    DO FOR all molecules in (half of) the neighboring cells
      IF both cluster criteria fulfilled
        get proper labels of both particles
        # ^ ^ important to do this here, can change in every step!
        link referring clusters to smaller proper label, sum size
      END IF
    END DO
  END DO
END DO
```

The previously described order of *(i)* do the linking within each cell and *(ii)* through the neighboring cells in the lattice; is not necessary. However it seems to be impossible to have real speed up at this spot. The checking has to be done for all molecules and cannot be stopped for a certain cell if one link is found, as it was possible in the method described before. This is a serious loss in performance.

Thinking about the communication the overall scheme will be conserved while the data that have to be exchanged will be different. The `cluster_id_border` array is handled exactly as it was done in the purely geometric cluster search. It is clear that also the positions and orientations have to be sent for every particle on the border. These are saved in `rxyz_border` again. To identify the entries of the `rxyz_border` array with the linking information in `cluster_id_border` also the former, globally unique label of every particle is important (see the initialization part of the pseudo code). This means there will be an additional array with 1 integer value for each molecule in the virtual bordering area of a processor, containing the original label. Obviously none of the sizes of these arrays is given in advance. So it is a good way to let the receiving processor guess the dimension of the original label array (which has the least amount of data inside) plus an uncertainty and receive an array of that size with the amount of data in the other arrays attached to the end. It would be possible to send the dimensions of the arrays in an additional communication step before transmitting them, however it seems that some overhead cannot be avoided.

Scaling

After all the theoretical discussions about a good implementation of the desired functionality it is now time to look at real time measurements. All measurements were carried out on the *ZAMpano* [11], a parallel computer with 8 compute nodes. One node consists of 4 processors and has 2GB shared memory. Therefore the memory model is only partly distributed. At first glance the communication tree should provide a *tree-like* scaling as it is described in [10]. That is the scaling is nearly linear for low numbers of processors and saturates at a constant value for large numbers. It does not show the behavior of *all-to-all* communication where for large processor numbers the performance is getting worse.

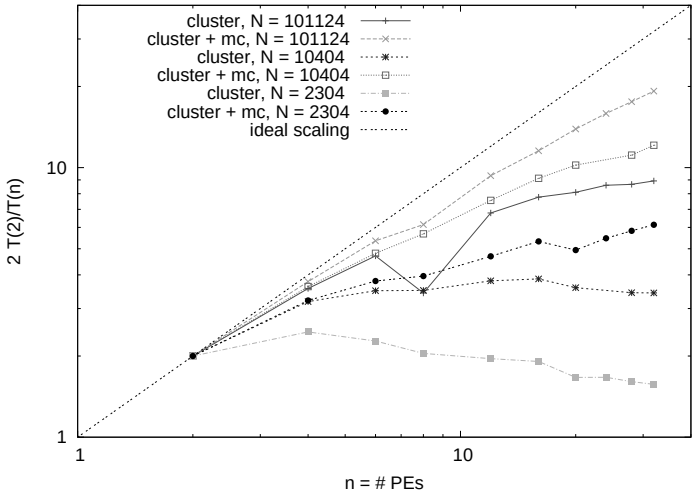


Figure 10: Scaling of the cluster algorithm itself and attached to the Monte Carlo step in *"SpinCG^{2d}"* for different system sizes and processor numbers.

Fig. 10 shows the measured scaling of the cluster routine itself and in combination with the *"SpinCG^{2d}"* program. Since the latter has an outstanding scaling behavior it is easy to accept that the combination of both always shows a better scaling compared to the pure cluster search. However it is found that the described technique scales more like an *all-to-all* communication scheme. Remembering fig. 8 and the conclusion that some kind of *all-to-all* communication is necessary this is not really astonishing. The reason for the difference of the *tree-like* communication and the scheme in the existing case is that the amount of data that has to be transmitted is also rising with the number of processors since the overall edge length is rising. This is not the case in the underlying eqn. 107 in [10]:

$$c(N_p) = \log_2(N_p)\lambda + \sum_{n=1}^{\log_2(N_p)} \frac{2^{n-1}\chi}{N_p}\,, \tag{6}$$

where λ is the latency and χ the bandwith. To have a better view on the real complexity with respect to N_p the sum can be simplified to:

$$c(N_p) = \log_2(N_p)\lambda + \left(1 - \frac{1}{N_p}\right)\chi \tag{7}$$

Here N_p is the number of processors, $c(N_p)$ expresses the relative portion of communication with respect to communication. In eqn. 6 the term $2^{n-1}/N_p$ is a normalized amount of data to be sent within each step. The overall amount is always 1, so in the last step of communication $1/2$ is sent. The edge length

is growing with about $\log_2(N_p)$ in two dimensions. So the leading behavior can be derived from 6:

$$c(N_p) = \log_2(N_p)\lambda + \sum_{n=1}^{\log_2(N_p)} \log_2(N_p)\frac{2^{n-1}\chi}{N_p}$$
$$= \log_2(N_p)\lambda + \log_2(N_p)\left(1 - \frac{1}{N_p}\right)\chi\,. \tag{8}$$

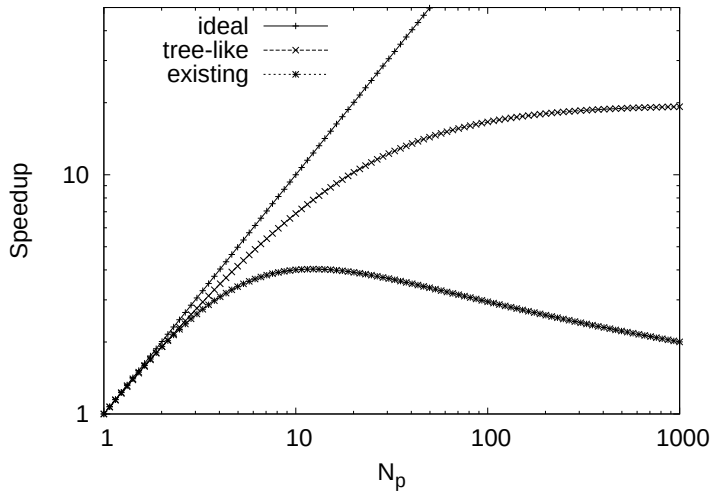


Figure 11: Comparison of *tree-like* scaling and that one which is theoretically expected for the communication scheme used in the cluster search algorithm for $\lambda = 10^{-4}$ and $\chi = 0.05$.

Since it cannot be avoided that the sum of the edges of all processors grows by increasing the number of nodes, it is at least satisfactory that after the slope of speedup got negative it is slowly increasing again (see fig. 11). The most important fact is that the good scaling behavior of *"SpinCG^{2d}"* is not destroyed by implementing the cluster algorithm into it. For that reason fig. 12 shows the absolute running times of different routines called in *"SpinCG^{2d}"* and the cluster labeling algorithm. The "local cluster" search is done absolutely independently on every processor and therefore it exhibits a good scaling behavior. The critical part of the *"SpinCG^{2d}"* concerning the number of processors is the "spatial decomposition" which contains communication. Its time consumption is more or less constant, independent of the number of PEs. The two parts of the cluster algorithm containing communication are fast enough not to make the negative scaling a real problem. The most time consuming but absolutely parallel "interaction" part is probably slowed down by the "spatial decomposition" at nearly the same number of processors where the global cluster identification becomes important, maybe this happens even earlier.

Cluster Size Distributions

Besides the important fact that the cluster search can extend the functionality of the Monte Carlo simulation, it is of course also possible to measure cluster size histograms or study percolation in final configurations. Due to the limited time only the distributions of cluster sizes in simulation of DNA molecules was studied. The measurement was done by creating 100 final configurations of a system with $N = 5184$ DNA strands by *"SpinCG^{2d}"* starting from a density $\rho = 2/\sqrt{3}d^2$, where $d = 35.0$Å was chosen. From final configurations an average distribution of cluster sizes was obtained. Since every configuration was obtained by a definite number of Monte Carlo steps originating from the same initial set-up (a hexagonal

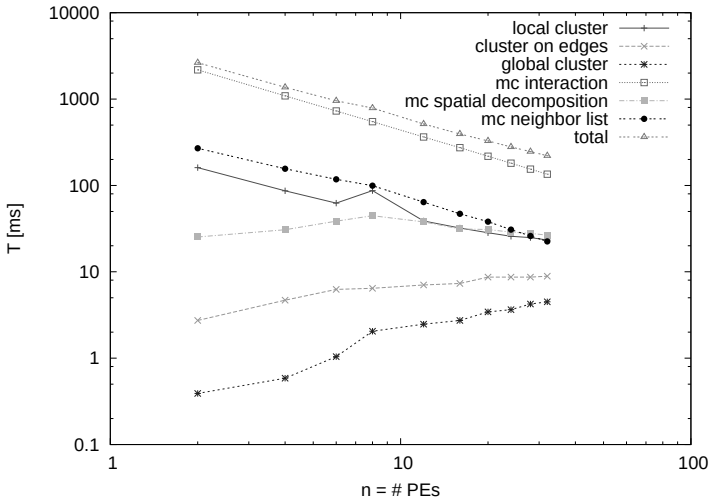


Figure 12: The runtime of different routines called in *"SpinCG^{2d}"* and necessary steps of the cluster search for various numbers of processors and a system size of about 100000 molecules.

lattice) the final systems were statistically absolutely independent and the error bars were calculated just as the standard deviation.

The measurements were carried out for different values of the charge compensation θ. The Debye screening length was always given by $\kappa^{-1} = 10$Å. As it was discussed in the beginning for this κ a phase transition is to be expected for $\theta \approx 0.86$. For smaller charge compensations the potential becomes completely repulsive which results in a kind of crystalline state of the system. For this reason there is no way to identify clusters, because it is even not possible to define a cluster criterion. Either all particles would be in one large cluster, or every particle would be identified to be separate from all others. Because of the repulsive form of the potential it is more reasonable to talk of a crystalline phase without any clusters. For higher values of the charge compensation θ the structure will be interesting because of the frustrations arising from the orientation dependent part of the Kornyshev-Leikin potential. It is interesting if there can be found some power law for the cluster size distribution in analogy to percolation on lattices for example.

Fig. 13.**a** shows a system with 324 DNA strands that was obtained by a simulation of 10000 Monte Carlo steps for $\theta = 0.95$. For this small number of molecules the picture is not capable to show the overall structure of such a system. But it depicts the qualitative fact that large clusters arise where the strands orient perpendicular to each other according to the minimum in the two dimensional potential energy landscape. Unlike in 13.**b**, where the same simulation was done with $\theta = 0.865$ close to the expected critical point. Most of the particles have a serious distance from each other while some of them gather into very small clusters at the flat minimum in the potential curve. This difference will be reflected in the cluster size histograms which allow qualitative statements about the structure of the system. In fig. 13.**c** and **d** the corresponding histograms to the structures in 13.**a** and **b** manifest this difference. For high charge compensation the probability of huge clusters is still in an acceptable range. Clusters at sizes greater than 2000 are likely to be percolating because they contain about half of all molecules in the system. The searched power law is well reproduced for long simulation times and the according exponent can be measured. For a rapid cooling and less Monte Carlo steps a deviation from the power law at a certain point can be observed. Obviously this can be taken as a sign for too short simulation times. Near to the critical point (fig. 13.**d**) the slope of the cluster size curve changes dramatically (see also fig. 14). It can also be seen that the deviation from a power law which could be observed for short simulation times in fig. 13.**c** appears even for 10000 Monte Carlo steps in this case. Therefore for the

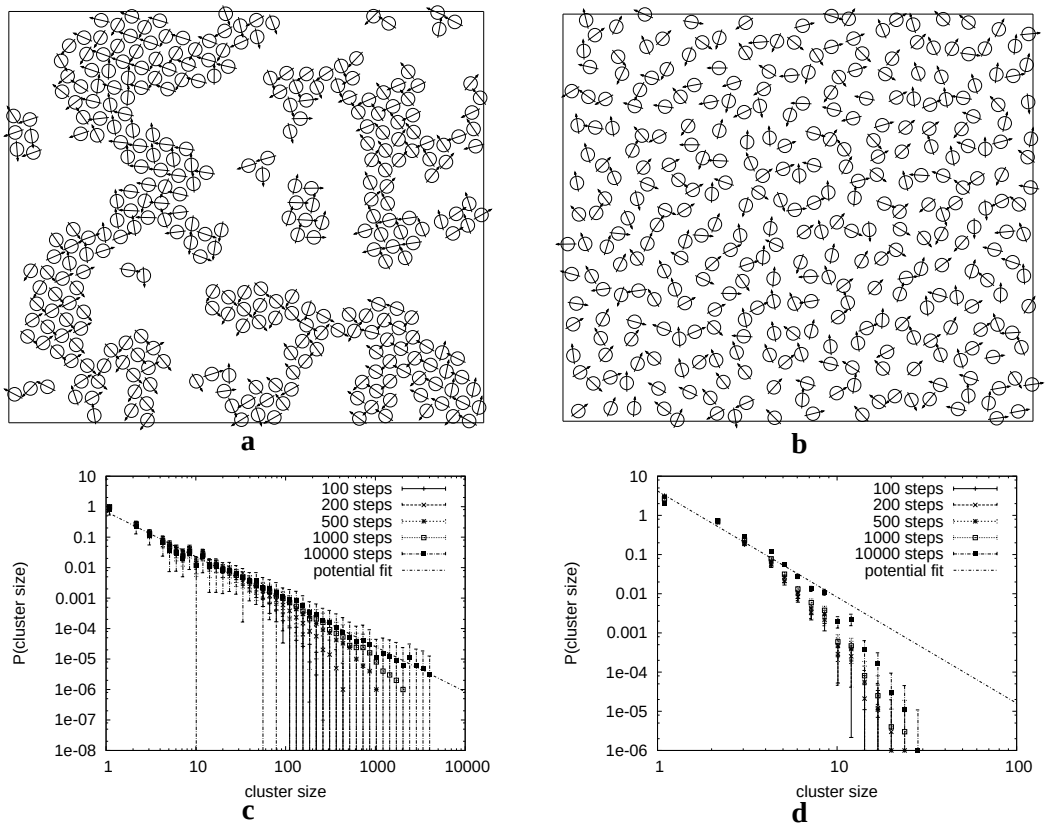


Figure 13: **a**, **b**: Final configuration of a model DNA aggregate with 324 molecules for $\theta = 0.95$ (**a**) and $\theta = 0.865$ (**b**) after simulated annealing over 10000 steps. The vectors show the azimuthal orientation of a molecule. **c**, **d**: The according cluster size histograms, averaged over 100 final configurations of a system with 5184 particles.

rough estimation of an exponent only the first values were taken into account. This behavior seems to be similar to the critical slowing down which often occurs at phase transitions in spin simulations. It would be interesting to see the effect of a real cluster algorithm in the Monte Carlo step. May be it would help to increase the quality of the data.

Finally in fig. 14 the exponents of the cluster size distributions are plotted for several θ. Although the error bars are only obtained from the least square fit (double standard deviation) the qualitative drop of the exponent close to $\theta = 0.86$ is obvious.

Conclusion and Outlook

The introduced cluster search algorithm can be used to build a Monte Carlo cluster algorithm for continuous two dimensional off-lattice spin systems on the one hand, and the study of cluster distributions and percolation on the other. It is designed for parallel, distributed memory systems and does not need to replicate memory. Although a variety of optimizations were implemented it cannot be avoided that the scaling of the method can have negative slopes. But since its absolute run time is very short it does not have any considerable influence on the good scaling behavior of *"SpinCG^{2d}"* and can therefore extend the functionality of the existing code.

It will be very interesting to implement an energetic criterion which could make the method capable of lowering the autocorrelation times in the described systems. Measurements of autocorrelation times

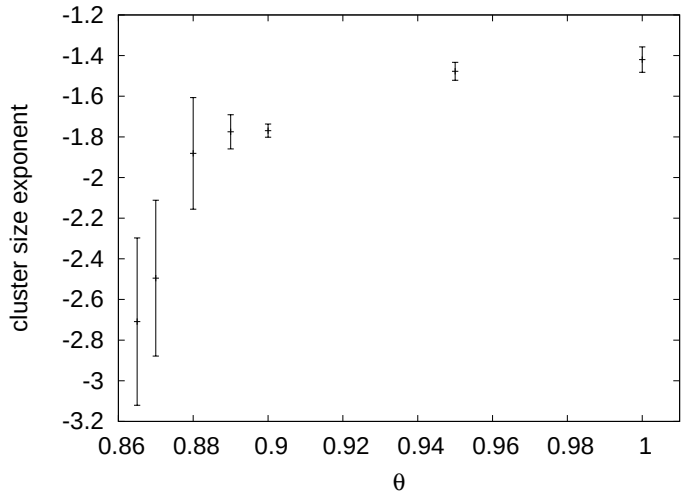


Figure 14: The rough exponent of the cluster size distributions for different values of θ.

in the current and extended version would be necessary to prove that. Also the measurements on system structure exponents should be carried out more thoroughly. An extension of the technique to three dimensions would be desirable and should not be complicated in principle.

Acknowledgments

I would like to thank Dr. Dr. Lippert and Dr. Esser for inviting me to the "Guest student program 2004" at the ZAM/NIC, Research Center Jülich, where this work was produced. Also I want to express my gratitude to my supervisor Dr. Godehard Sutmann, who did not only introduce me to a very interesting topic of research but also assisted me with the development, implementation, analysis and application of the described algorithm. I am grateful to the employees of the ZAM whose engagement was essential for the great experience I had here in Jülich, especially Jörg Striegnitz who taught us the basics of parallel programming.

References

1. A. A. Kornyshev and S. Leikin, J. Chem. Phys. **107**, 3656 (1997)
2. G. Sutmann, Monte Carlo Simulations of Columnar DNA Aggregates, to be published
3. G. Sutmann, Temperature induced structural transitions in columnar DNA aggregates, to be published
4. A. A. Kornyshev and S. Leikin, Proc. Natl. Acad. Sci. U.S.A. **95**, 13597 (1998)
5. H. M. Harreis, A. A. Kornyshev, C. N. Likos, H. Löwen, G. Sutmann, Phys. Rev. Lett. **89**, 018303-1 (2002)
6. G. Sutmann, *SpinCG^{2d} - a parallel Monte Carlo program for spin systems*, in preparation
7. J. Hoshen, R. Kopelman, Phys. Rev. B **14**, 3438 (1976)
8. R. H. Swendsen, J. S. Wang, Phys. Rev. Lett. **58**, 86 (1986)
9. U. Wolff, Phys. Rev. Lett. **62**, 361 (1988)
10. G. Sutmann, *Classical Molecular Dynamics*, in *Quantum Simulations of Complex Many-Body Systems: From Theory to Algorithms*, Lecture Notes, J. Grotendorst, D. Marx, A. Muramatsu (Eds.), John von Neumann Institute for Computing, Jülich, 2002, Vol. 10, p. 211-254.
11. http://zampano.zam.kfa-juelich.de/

Resource Negotiation in Unicore: The Web Services Agreement Approach

Jiulong Shan

National High Performance Computing Center (Hefei, China)
University of Science and Technology of China
Hefei Anhui, PR China, 230027

E-mail: jlshan@mail.ustc.edu.cn

Abstract: UNICORE, a popular Grid Computing software, facilitates the seamless and secure sharing of computing resources among distributed supercomputers. In its current implementation, UNICORE adopts a static resource management model which can not fully meet the user's requirement on resource provisioning. In this work, we propose the incorporation of a Web Services (WS) Agreement based resource negotiation model to advance the current resource management model of UNICORE. The feasibility of the proposed approach is validated through experiments based on a prototype implementation.

Introduction

Grids [1], geographically distributed computing platforms, aim at resource sharing and problem solving among heterogeneous, potentially large-scale resources. According to the underlying resource types, Grids can be further classified into Computing Grids and Data Grids. Computing Grids aim at the sharing of computing resources among supercomputers deployed among different HPC Centers while Data Grids aim at synthesizing new information from data repositories distributed in a wide area network.

Among all the Grid software, UNICORE [2] and Globus [3] are the two most important ones. Globus is an open source project that intends to provide Grid building tools for a Grid infrastructure. These tools include a set of core services, a set of advanced services and a set of well-defined APIs for Grid application construction. UNICORE, on the other hand, currently provides a vertically integrated solution for combining computing resources on the Internet. On the basis of a uniform access to distributed computing resources, it supports a powerful workflow management and a friendly client side user interface. UNICORE has recently been released as open source software as well.

While the resource management functionality of UNICORE is rich, in its current implementation it adopts a static resource management approach which can not fully meet the user's requirements on resource provision. This motivates the main goal of this work: to support dynamic resource negotiation in UNICORE.

The paper is organized as follows: The next section discusses the current resource management model in UNICORE and its weak points. Section 3 introduces the architecture of WS Agreement based resource negotiation. The implementation is shown in section 4. Finally, section 5 concludes the discussion and outlines future work.

Current Resource Management in UNICORE

UNICORE is the acronym of **Uni**form **I**nterface to **Co**mputing **Re**sources [4]. As shown in Fig. 1, the main components of UNICORE are Gateway, Network Job Superviser (NJS), Target System Interface (TSI) and Client.

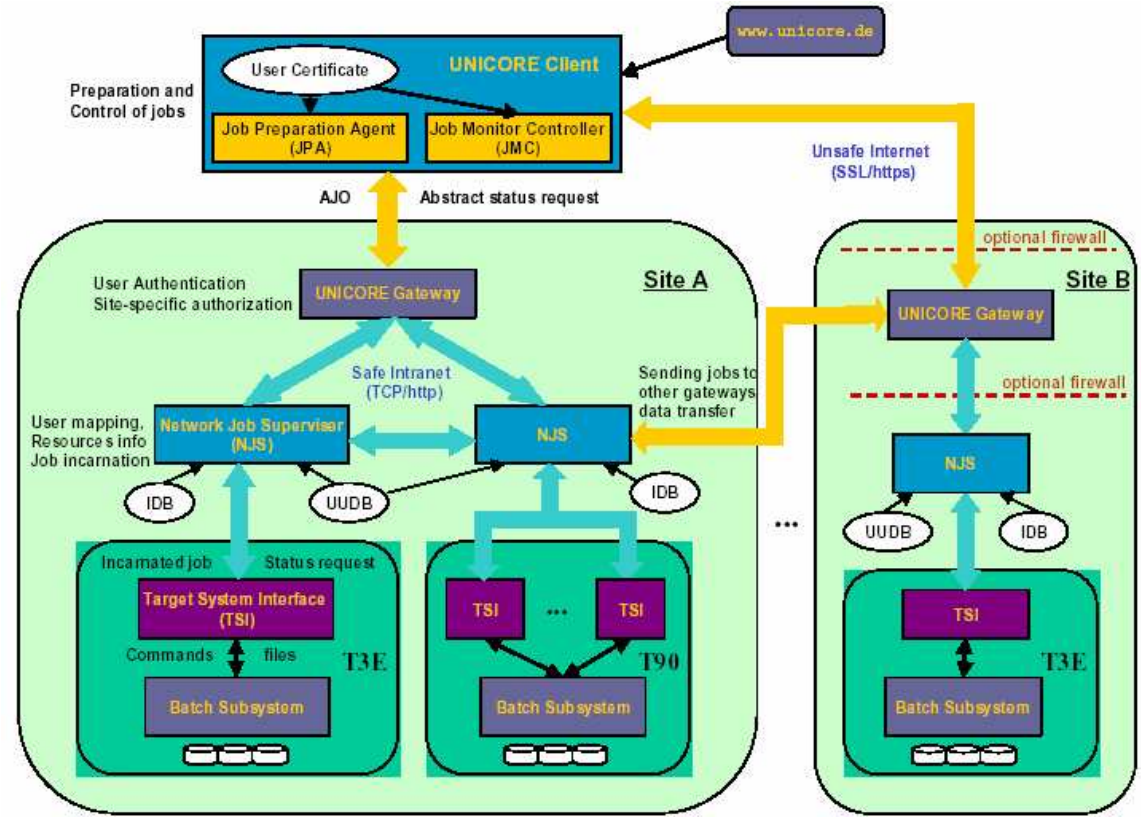


Figure 1: UNICORE System Architecture

TSI provides an interface between the abstract resource model of the NJS and the target supercomputer or storage server, through which NJS can submit jobs to the target system. These two parts establish a virtual site in UNICORE (Vsite). By connecting to a Gateway that is common for a single administrative domain, several Vsites build a UNICORE Grid site (Usite). The end-user uses the UNICORE client to connect to a Usite Gateway and control his job.

When preparing a job, end-users should first have the information of the available resources (Vsites) and then they can select the appropriate destination to run their job. In the current static resource management model, this information is managed through the following steps:

1. The resource information of a target system are configured in the static text Incarnation DataBase (IDB) file on NJS, such as the listed PROCESSOR information means there are totally 10 processors per Node and the request number must be between 1 and 10.

   ```
   PROCESSOR [Number of PEs per Node] DEFAULT [1] MAXIMUM [10] MINIMUM [1]
   ```

2. In the initialization process, the NJS reads this information into a static member variable ***resources*** and keeps it unchanged during its lifetime.

3. When connected to a Usite, the Client submits a *GetResourceDescription* action[1] to the NJS. The corresponding handler *DoGetResources* within the NJS reads the information from the static variable *resources* and returns it back to the Client.

4. After obtaining the resource information, end-users can make a match between their jobs and the resources. Finally, the end-users will submit their jobs to a Usite with a specified resource requirement.

Although this model can help the end-user to perform their job submission, it only provides a static resource view for a VSite and can thus not fully meet the requirements of both parties, the end-user and the provider, on resource provisioning.

First, end-users can not obtain the up-to-date resource information of a target system. For example the current system load, the free processor number, or the available memory size. Surely, this information affects the end-user to make a suitable resource selection. Second, the current model doesn't support the policy based resource provisioning as demanded by providers. For example the target system can not provide specified resource information according to the userid and it's the same for other policies. Third, the system can not provide a QoS ensured resource provisioning. It only provides the resource information and transfers the job to the selected target system. There are no intermediate assurance components. Finally, economic models of resource provisioning [5] will play an important role in Grid Computing. But in the current model, it is hardly possible to implement this.

To meet the above requirements, we introduce a negotiation based resource management model into UNICORE.

WS Agreement based Resource Negotiation in UNICORE

Currently, there are several Service Level Agreement (SLA) protocols that can support resource negotiation in distributed systems, such as WSLA [6], BPEL [7] and WS Agreement [8]. Among them, WS Agreement is a newly proposed and WSRF based protocol.

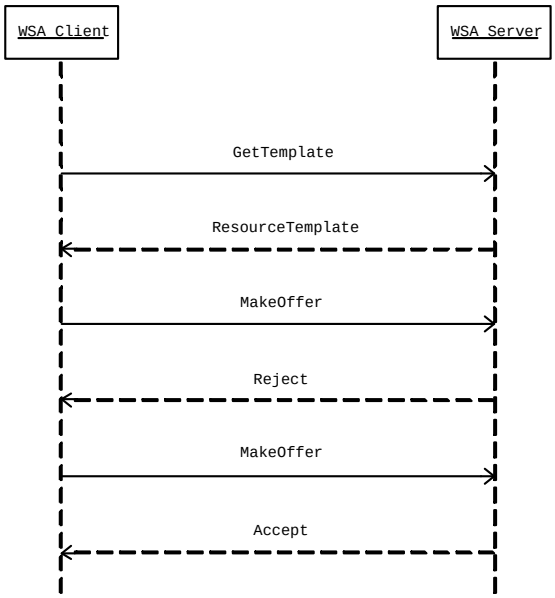


Figure 2: Interaction Schema of WS Agreement

[1] See Appendix A

Web Services Agreement

WS Agreement describes an XML language for the specification of an agreement between a resource provider and a consumer. WS Agreement also specifies a protocol for the creation of an agreement using agreement templates. The specification includes not only the scheme for specifying an agreement but also a set of port types and operations for managing the agreement.

Fig. 2 shows the basic interaction scheme of WS Agreement. First, the consumer needs to request for a resource template from the provider and after several negotiation steps there will be an agreement between these two parties. In the specification, an EPR will be returned to the WS Agreement Client as a unique identification of the created agreement, which is used for further agreement managing.

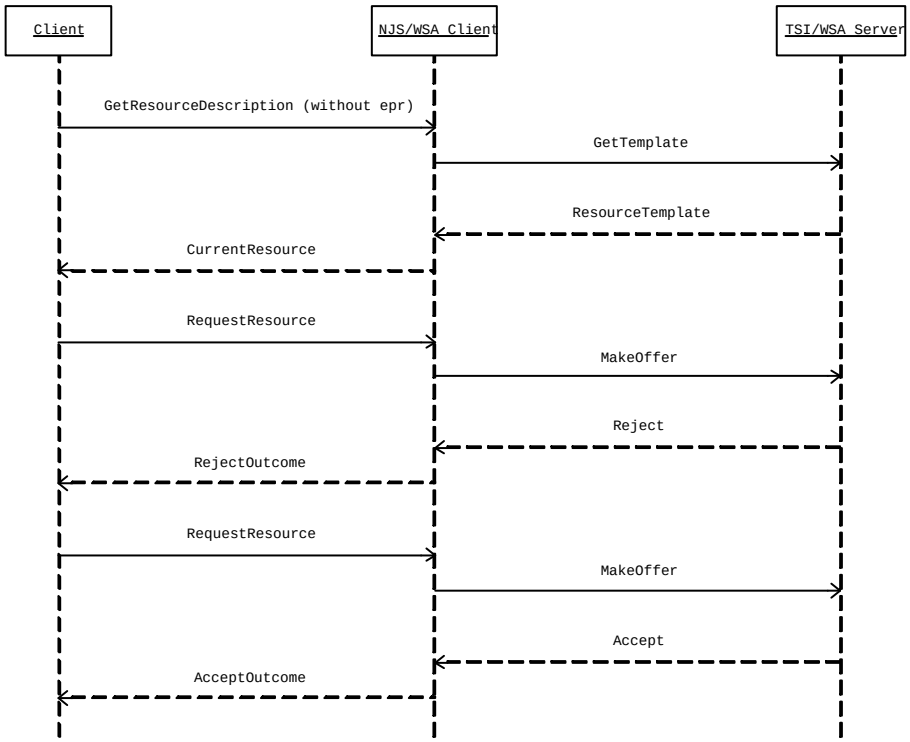


Figure 3: Interaction Scheme of UNICORE

System Architecture of Resource Negotiation Model

According to the WS Agreement specification and the requirements for resource provision, we designed the interaction schemes for the resource negotiation of UNICORE. As shown in Fig. 3, the approach follows that of WS Agreement. However, within the UNICORE architecture the NJS communicates with the WS Agreement Server on behalf of the end-user.

The system architecture for this resource negotiation model of UNICORE is shown in Fig. 4. Compared with the original model, the resource information is managed as follows:

1. The same as the original model, NJS reads the resource information into a static member variable at initialization. But it's only taken as a reference for future use.
2. The Client can make a *GetResourceDescription* request to NJS. Acting as a representative of the end-user and also as a WS Agreement Client, NJS will request the resource template, make an agreement offer, and query the agreement properties according to the contents of this request.

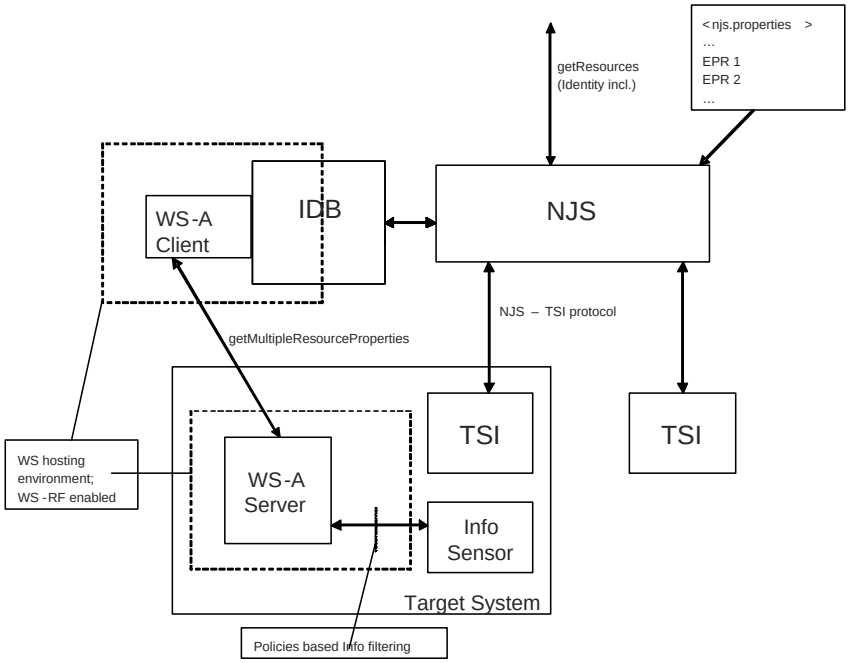


Figure 4: Architecture for Resource Negotiation in UNICORE

(a) If the request doesn't contain any attached information, NJS will send *GetTemplate* to the WS Agreement Server in order to get dynamic and user specified resource information. This information will be returned to the end-user through the Gateway.

(b) If the request contains a *ResourceSet*, the NJS will treat it as a resource request action and will send a *MakeOffer* message to the WS Agreement Server to create an agreement. When an agreement is successfully created, the corresponding End-Point-Reference (EPR) will be sent back to the end-user.

(c) If the request contains a valid EPR, it means that this is a resource query action. The NJS will use this EPR to query multiple properties of the corresponding agreement and returns the information to the end-user.

3. With the returned EPR, an end-user can make full use of this resource agreement.

With all the above functions, the new negotiation based resource management model can surely meet the requirements described in section 2.

System Implementation and Demos

Based on the architecture shown in Fig. 4, we implemented a prototype system. In the implementation, we modified NJS to support the new resource management model. Also, we integrated the new UNICORE version with WS Agreement. After that, we modified the Client of UNICORE to validate the feasibility of this new model. In addition to the negotiation requirements, the compatibility with the original system was another requirement of this work.

Modification of NJS

As described in Appendix A, each request from the Client can be seen as an action. In this framework, there are two ways to do the modification:

First, we can add some new actions into the Abstract Job Object (AJO) to support resource negotiation. While this approach is relatively easy to perform and without changing the original system design principle, the modification of AJO will affect almost every part of UNICORE, which means losing of compatibility.

Second, by modifying the existing actions we can also achieve the same goal. Although it will disobey the design principle of UNICORE, this method will limit the modification inside NJS and can meet the compatibility requirement.

In the prototype implementation, we chose the latter one and modified the handler for the action *GetResourceDescription*. The pseudo-code is listed below:

```
String globalInfo = this.getUspace().getDirectory() + ".UNICORE_GLOBAL_INFO";
if ((new File(globalInfo)).exists()) {
    ResSet = read ResoruceSet from File globalInfo;
    ag_epr = read resource from ResSet;

    // If there is a valid EPR then this is a resource query Request
    // else this is an make offer request
    if (ag_epr == null) {
        ERP = make an agreement offer;
        return EPR;
    } else {
        newResSet = query resource properties;
        return newResSet;
    }
} else {
    // This is a get template request
    newResSet = get resource template;
    return newResSet;
}
```

Besides this, we also modified the task management components of NJS to check whether the incoming job has a valid resource agreement.

Integration of WS Agreement

WS Agreement is an developing specification. We used a prototype implementation [9] based on the specification version 1.1, draft 20.

By introducing the class *DoGetDynamicResInfo*, we can also use NJS as a WS Agreement Client. Class *GenerateResourceOffer* and *ParseResourceTemplate* are used to deal with the resource template and agreement offer. The pseudo-code is listed below:

```
public class DoGetDynamicResInfo {
    /**
    * Read the Dynamic Resources Info form WS Agreement by GetTemplate,
    * the EPR here doesn't contain the agreement information but for the
    * WS Agreement Server
    */
    public static ResourceSet getDynamicResource(String xlogin, String vsite_epr,
                                                 ResourceSet current_rs) {
    }

    /**
    * Read the Dynamic Resources Info form WS Agreement by GetMultiProperties,
```

```
        * here the EPR stands for the agreement
        */
        public static ResourceSet getDynamicResource(String vsite_epr, String ag_epr) {
        }
    }
```

Modification of Client

In order to validate the implementation of a new resource management model, it was needed to create a client for communicating with NJS. There are also two ways to achieve this:

First, we can use the Arcon Client of UNICORE to test this implementation. The Arcon Client is a library that provides a simple, lightweight interface to the UNICORE servers and can be used to build UNICORE clients. It is easy for programming, but it only provides a limited set of functions and does not have a graphic user interface.

Second, we can modify the normal UNICORE Client to add new functions. The normal client has rich functions and a well designed graphic user interface. But the complex architecture makes it difficult for modification.

In this work, we chose the second one. Below is a list of classes that we modified or added: *Client*, *ResourceManager*, *GetResources*, *JobGroupPanel*, *JobResourcePanel*, *ResourceQueryPanel*, *ResourceQueryDialog*, *JMCTree*, *JPATree*.

Demos

By using the modified Client we can get the up-to-date resource information, create an agreement and can query the resource properties of this agreement. Some screen captures are collected in Fig. 5.

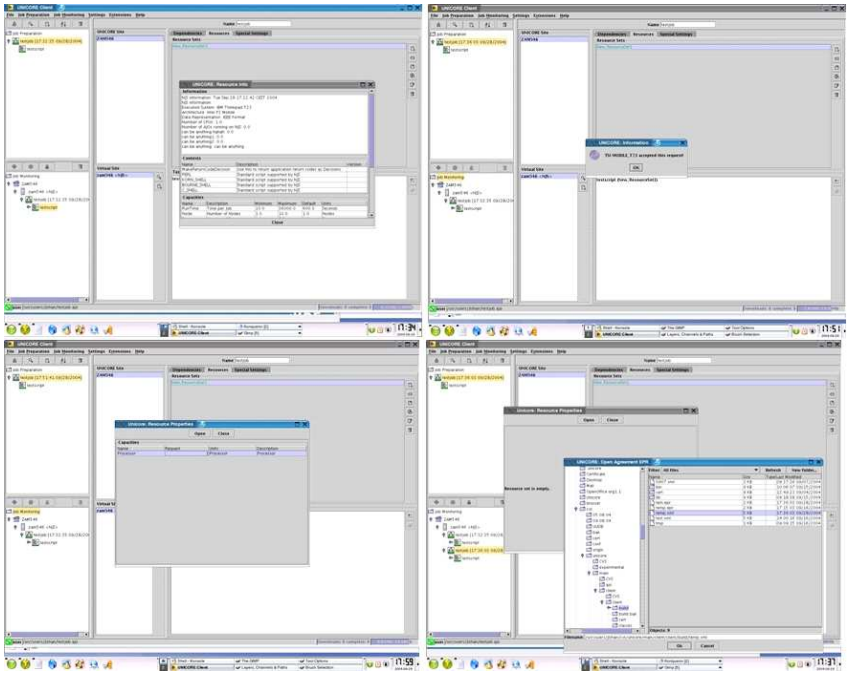


Figure 5: Demos of the Negotiation based Resource Management Model

Conclusion and Future Work

In this work, we identified some weak points of the current resource management model in UNICORE and proposed the negotiation based model as a solution. Through the prototype implementation and a set of experiments, we demonstrated the feasibility of this new model.

With the exploration of this work, this model can be easily adopted by the next version of WSRF based UNICORE software. Furthermore, on top of this model, we can also achieve resource scheduling and work flow management to provide a better resource provisioning.

References

1. I. Foster, C. Kesselmann, editors, The Grid: Blueprint for a New Computing Infrastructure, Morgan Kaufman Publishers, 1998.
2. M. Romberg, The UNICORE Architecture: Seamless Access to Distributed Resources, International Symposium on High Performance Distributed Computing (HPDC-8), 1999.
3. I. Foster, C. Kesselman, Globus: A Metacomputing Infrastructure Toolkit, International Journal of Supercomputer Applications and High Performance Computing, 1997.
4. D. Erwin, ed., UNICORE plus final report – uniform interface to computing resources, Forschungszentrum Jülich, 2003.
5. R. Buyya, H. Stockinger, et al., Economic Models for Management of Resources in Peer-to-Peer and Grid Computing, SPIE International Symposium on The Convergence of Information Technologies and Communications (ITCom 2001), 2001.
6. IBM Cooperation, Web Service Level Agreements Projects, http://www.research.ibm.com/wsla/ 2001-2003.
7. IBM Cooperation, Business Process Execution Language for Web Services Version 1.1, http://www-106.ibm.com/developerworks/library/ws-bpel/ 2003.
8. A. Andrieux, K.Czajkowski, A. Dan, et al., Web Services Agreement Specification, Version 1.1, Draft 20, 2004.
9. M. Riedel, Prospects and Realization of Flexible Service Offers in a Grid Environment, Diploma Thesis, 2004.

Appendix

Resource Information Management in NJS

The following descriptions are mainly concentrated on where and how the resource information is managed in NJS, including initialization and action handling.

Initialization of NJS

As shown in Fig. 6, class *NJS* is in charge of the initialization phase and the whole process is listed as follwos:

1. Class *NJS* gets the name of property file (njs.properties) and some other parameters from startup script and then passes them to class *Configuration*.

2. Class *Configuration* reads all the predefined properties for NJS from the property file and stores them into the static member variables of class *NJSGlobal*. These properties in class *NJSGlobal* will be used by other parts of NJS.

3. Class *NJS* calls the function *initialise()* of class *Seminaries* to read the static resource information from IDB file. When finished parsing IDB file, the resource information is stored in the static member variable *resources* of class *DoGetResources*.

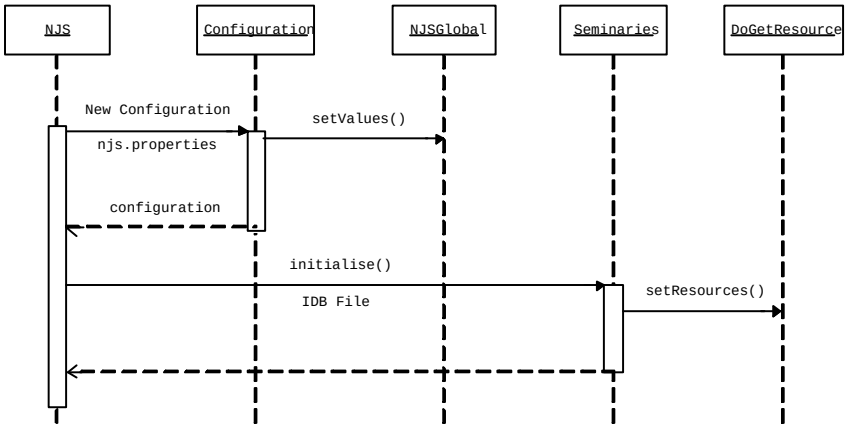


Figure 6: Initialization of NJS

In NJS, there are two kinds of objects working together for parsing the resource information: Missal and Reader. Missal contains a set of raw resource information and a dictionary of different kinds of resource readers. For example, class *ResourceReader* contains the readers for TextInfo Resource, NumericInfo Resource, Capacity Resource, and class *TargetSystem* contains the readers for Software Resource.

In initialization, the content of IDB file will be read into a Missal object first. And then this Missal object can parse the resource information by using different resource readers.

Action Handling in NJS

In UNICORE each kind of request is taken as an action and there are four kinds of actions defined in AJO: RAction, XAction, EAction and NAction. NAction refers to those requests that should be executed by NJS itself.

For each action there is the corresponding KnownActionFactory and Handler, such as *GetResourceDescription*, *DoGetResources.Factory* and *DoGetResources*. In UNICORE *GetResourceDescription* is a NAction responsible for providing the static resource information to the end-user.

In the initialization of NJS, class *NJS* calls the *init()* function of class *KnownActionFactory* to initialize the relationship between Action and Factory, such as the *GetResourceDescription* action has the following registration sentences:

```
classes_e.add((new GetResourceDescription()).getClass().getName());
handlers_e.add(new DoGetResources.Factory());
```

With the help of this relation KnownActionDB, NJS can first create a handler instance and then dispatch the action to it for processing.

Acknowledgements: I would like to express my gratitude to many people of ZAM who helped me a lot in the past two months. Dr. Ruediger Esser gave me this valuable chance to participate in this Summer School Program; Dr. Volker Sander guided my research work with precious ideas, suggestions and comments; Dietmar Erwin provided me with this favorable working environment; the discussions with Philipp Wieder, Roger Menday, Bernd Schuller and Mathilde Romberg helped me on the way to the soul of the UNICORE project; Morris Riedel accompanied along the whole project on both technique and life; the time spent with all the guest students in this program is memorable.

Leistungsanalyse der Matrixmultiplikationsroutinen PDGEMM / PDSYMM auf IBM p-690 (Jump)

Benjamin Sohn

Technische Universität Dresden
Fakultät Mathematik und Naturwissenschaften
Fachrichtung Mathematik

E-mail: sohn@math.tu-dresden.de

Zusammenfassung: Die Performance der parallelen Matrix-Matrix-Multiplikationsroutinen PDGEMM und PDSYMM wurden auf dem JUelich Multi Processor Jump [1] untersucht.

Einführung

Untersucht wurde die Routine PDGEMM (P für Parallel, D für Double precision, GE für die Matrix-Eigenschaft GEneral und MM für Matrix $\times$ Matrix) aus PESSL, die die folgende Operation durchführt:

$$C \leftarrow \alpha op(A) op(B) + \beta C, \quad op(X) = X, X^T \quad \text{mit } A \in \mathbb{R}^{m \times k},\ B \in \mathbb{R}^{k \times n},\ C \in \mathbb{R}^{m \times n}$$

Des weiteren wurde die Routine PDSYMM (P für Parallel, D für Double precision, SY für die Matrix-Eigenschaft SYmmetric und MM für Matrix $\times$ Matrix) aus PESSL untersucht, die die nachfolgende Operation ausführt:

$$C \leftarrow \alpha AB + \beta C, \quad C \leftarrow \alpha BA + \beta C \quad \text{mit } A^T = A \in \mathbb{R}^{m \times m}$$

Diese Routine benötigt keine vollständige Matrix, sondern nur eine obere bzw. untere Dreiecksmatrix, der Rest dieser Matrix kann beliebig besetzt sein. In diesem Dokument wurde der Einfachheit halber nur mit quadratischen Matrizen gerechnet, also $k = m = n$. Es wurden folgende Werte zur Berechnung benutzt: $\alpha = 1, \quad \beta = 1$. Die Matrizen wurden auf folgende Weise erzeugt: $A(i,j) = B(i,j) = (i+j)/100$ und, sofern nicht anders bezeichnet, wurden alle Messungen mit der Multiplikation zweier 1000×1000-Matrizen durchgeführt.

Bei der Verteilung der Matrizen auf die einzelnen Prozessoren (siehe auf Seite 105) muss noch gesagt werden handelt es sich um eine block-zyklische Verteilung (siehe ScaLAPACK User's Guide [2]). Dies bedeutet, dass zunächst die Matrix in Blöcke aufgeteilt wird. Dann wird das Prozessor-Gitter immer wieder über die Matrix mit ihren einzelnen Blöcken gelegt. Dadurch entstehen unter Umständen Last-Ungleichgewichte, da nicht alle Prozessoren dieselbe Menge zur Berechnung erhalten. Näheres wird in dem dazugehörigen Kapitel auf Seite 105 erklärt und veranschaulicht.

Der JUelich Multi Processor Jump besteht aus 41 IBM p690 Knoten, wobei jeder Knoten mit 32 Power4+ Prozessoren mit 1.7 GHz Taktung bestückt ist, was einer Gesamt-Anzahl von 1312 Prozessoren entspricht. Die Leistung pro Prozessor beträgt 6.8 GFlops, woraus sich eine Gesamtleistung von 8.9 TFLOPS ergibt. Jeder Prozessor verfügt über 64 / 32 KB instruction / data internen Level1 Cache. Je zwei Prozessoren teilen sich 1.5 MB Level2 Cache und können auf 512 MB Level3 Cache pro Knoten zugreifen. Jeder Knoten verfügt des weiteren über 512 GB Speicher. Die Berechnungen wurden unter dem Betriebssystem AIX 5.2 mit ESSL V4.1 [3], PESSL V3.1 [4] und dem Fortran Compiler XL Fortran 8.1 durchgeführt. Die Zeiten wurden mit der Routine MPI_WTIME gemessen.

Die Messungen wurden alle auf ganzen Knoten durchgeführt. Die Option @node_usage = not_shared im LoadLeveler bewirkte, dass keine anderen Benutzer die noch freien Prozessoren nutzten, um eventuellen Kommunikations- oder Speicher-Konflikten aus dem Weg zu gehen. Da man es gerade bei kleineren Matrizen mit extrem kurzen Zeitspannen zu tun hat, wurden die Messungen mehrfach wiederholt und die Ergebnisse gemittelt. In den folgenden Graphiken werden oft die MFLOPS angegeben. Diese wurden anhand der Annahme, dass eine Multiplikation zweier $n \times n$ - Matrizen $2n^3$ $Floating\ Point$ Operationen benötigt, berechnet.
Ein weiteres Problem stellt das Betriebssystem dar, da in regelmäßigen Abständen bestimmte Dienste aufgerufen werden, die die Zeitmessungen verfälschen. Die Zeit läuft weiter, obwohl eigentlich keine für die Berechnung relevanten Ausführungen getätigt werden. Dies ist durch Ausreißer in den Grafiken zu erkennen.

Performance von PDGEMM

Bei der Nutzung der parallelen Matrixmultiplikationsroutinen PDGEMM und PDSYMM müssen, um eine maximale Leistungs-Ausnutzung des Computers erreichen zu können, mehrere Parameter beachtet werden: Prozessor-Anordnung, Prozessor-Anzahl und Blockgröße.

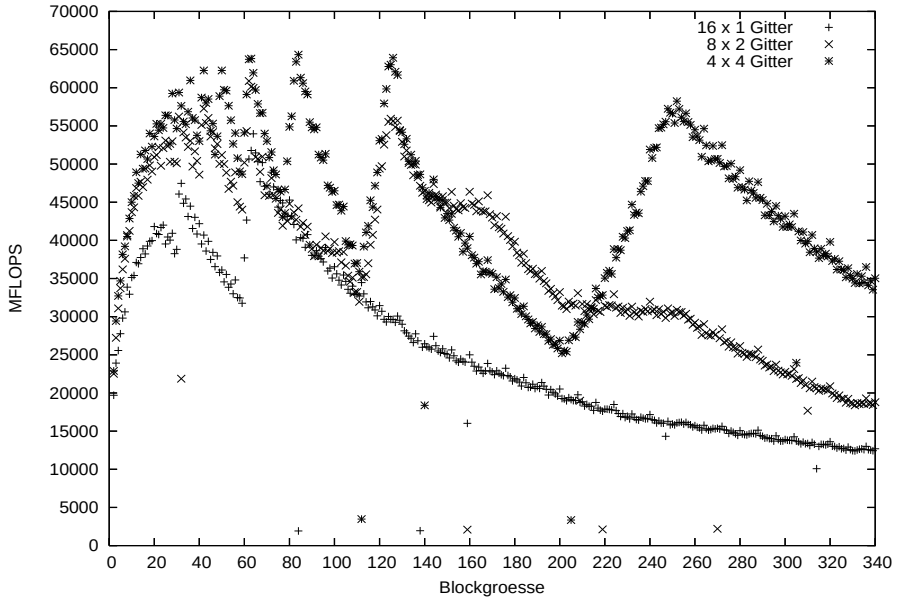


Abbildung 1: Performance von PDGEMM: unterschiedliche Blockgrößen, verschiedene Prozessoranordnungen

In Abbildung 1 auf der vorherigen Seite erkennt man die starke Abhängigkeit der Performance von Blockgröße und Prozessoranordnung, wobei die Performance auf nichtquadratischen Gittern bei größeren Blockgrößen stark zurück geht, da nicht mehr alle Prozessoren arbeiten.

Wenn einer der oben genannten Parameter ungünstig gewählt wird, muss man mit starken Performance-Einbußen rechnen. Im folgenden werden die einzelnen Parameter und die damit verbundenen Probleme näher erläutert.

Die Prozessor-Anordnung

Nicht nur die eigentliche Anzahl der Prozessoren (siehe „Die Prozessoranzahl“ auf der nächsten Seite), sondern auch deren Anordnung spielt für eine optimale Performance ein große Rolle. Dies hängt damit zusammen, dass die Prozessoren bei einer gut verteilten Anordnung bessere Kommunikationsmöglichkeiten haben, da dann mehrere Prozessor-Spalten oder -Zeilen parallel miteinander kommunizieren können und nicht nur eine globale Kommunikation abläuft. Wie in Abbildung 1 zu erkennen ist, erhält man bei einem 16×1 Gitter eine akzeptable Performance bei kleineren Blockgrößen (siehe „Die Blockgröße“ auf Seite 105), danach bricht die Performance ein. Bei dieser Anordnung liegen alle 16 Prozessoren in einer Reihe und es ist nur eine einzige Kommunikation realisierbar. Die beste Performance bei diesem Gitter beträgt 53963 MFLOPS. Im Vergleich dazu beträgt die höchste Performance bei einem 4×4 Gitter 63929 MFLOPS, also circa 18 Prozent mehr. In diesem Gitter liegen die Prozessoren optimal verteilt und können so besser miteinander kommunizieren.

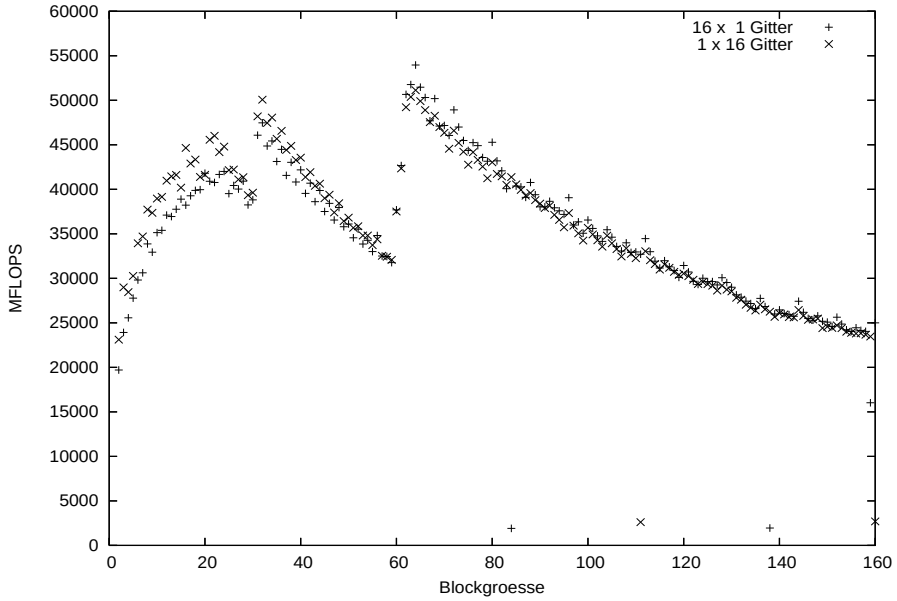


Abbildung 2: Performance von PDGEMM: unterschiedliche Blockgrößen, verschiedene Prozessoranordnungen (horizontal und vertikal)

Abbildung 2 zeigt, dass es auch Unterschiede macht in welcher Richtung die Gitter angeordnet sind, dies lässt sich auf die unterschiedlich Ausnutzung des Caches zurückführen, denn Fortran speichert Matrizen immer spaltenweise ab. In dieser Abbildung werden bereits ab einer Blockgröße von 67 nicht mehr alle Prozessoren beansprucht.

Die Prozessor-Anzahl

Es ist wichtig eine „geschickte“ Anzahl von Prozessoren auszuwählen, denn diese ist eng mit der Effektivität verbunden. Der Speedup von PDGEMM ist durch die Kommunikation nach oben begrenzt, da bei mehr Prozessoren, zwar der Anteil an der sequenziellen Berechnung auf jedem Prozessor geringer wird, aber gleichzeitig der Kommunikations-Aufwand steigt.

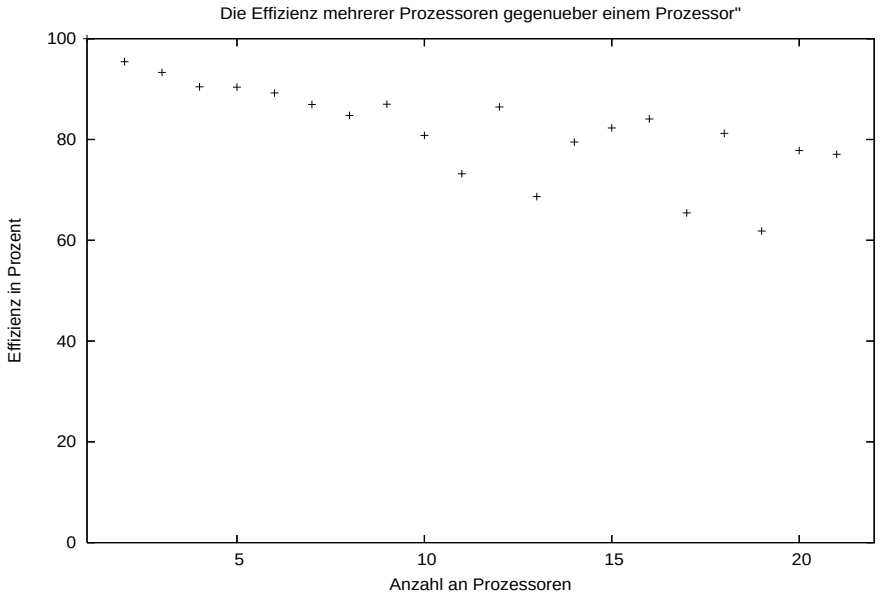


Abbildung 3: Effektivität von PDGEMM: verschiedene Prozessoranzahlen bei optimalem Gitter und optimaler Blockgröße

In Abbildung 3 kann man sehen, wie die Effektivität (1 Prozessor = 100 Prozent) bei steigender Anzahl von Prozessoren immer weiter nachlässt. Des weiteren sieht man, dass Primzahlen als Anzahl von Prozessoren vollkommen ungeeignet sind, was damit zusammenhängt, dass die Prozessoren entweder in einer Reihe oder in einer Spalte angeordnet und nicht gleichmäßig verteilt sind, wohingegen mit einer Prozessor-Anzahl die gut teilbar ist, deutlich bessere Ergebnisse zu erzielen sind - am Besten ist eine quadratische Anzahl, da die Prozessoren dann ideal aufgeteilt sind.

Prozessoren	1	2	4	8	16	32	64	128
Zeit in s	.42079	.22047	.11630	.06206	.03128	.01883	.01162	.01007
Speedup	1	1.91	3.62	6.78	13.45	22.35	36.22	41.78

Tabelle 1: Speedups bei Multiplikation von 1000×1000 Matrizen

Tabelle 1 zeigt den Speedup bei fester Matrixgröße und Verdopplung der Prozessorzahlen. Man sieht, dass man in der Realität niemals 128 Prozessoren für eine so geringe Matrixgröße nutzen würde, da jeder einzelne Prozessor bei einer 1000×1000-Matrix nur noch sehr kleine Blöcke zur Berechnung erhält. Bei größeren Matrizen wäre jedoch der Speedup auch mit mehr Prozessoren noch akzeptabel.

Man benötigt, um die Zeiten zu halbieren, eine immer größere Anzahl an Prozessoren, dadurch muss man abschätzen, bis wohin sich eine Beschleunigung noch lohnt und ab wann man eher wieder auf

weniger Prozessoren zurückgreift, um Ressourcen zu sparen. Der Vergleich zwischen 36 und 72 Prozessoren in Tabelle auf Seite 120 zeigt dies deutlich, trotz Verdopplung der Prozessor-Anzahl beträgt der Zeitgewinn gerade noch 26 %.

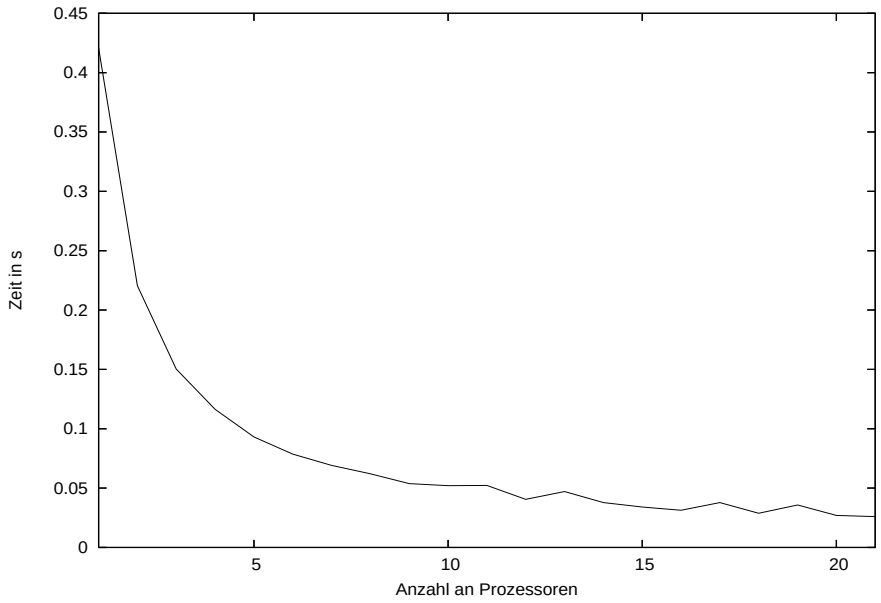


Abbildung 4: Dauer in Sekunden für Berechnung mit verschiedener Prozessoranzahl (optimales Gitter / optimale Blockgröße)

Abbildung 4 veranschaulicht, wie der Nutzen von mehr Prozessoren immer geringer wird, und die Ausführungszeit teilweise sogar wieder ansteigt, wenn eine ungünstige Anzahl an Prozessoren gewählt wurde.

Die Blockgröße

Die Blockgröße ist für die eigentliche Performance der alles entscheidende Faktor, denn diese Größe gibt an, wie groß die Blöcke sind, die jedem Prozessor zugeteilt werden. Dabei sind größere Blöcke kleineren vorzuziehen.

Sind diese Blöcke ungünstig gewählt, so kann es passieren, dass ein Ungleichgewicht bei der Berechnung entsteht - ein Prozessor muss bedeutend mehr berechnen, als ein anderer. Im Extremfall kann es bei viel zu groß gewählten Blöcken sogar passieren, dass einige Prozessoren nichts mehr berechnen und die Last auf die anderen fällt.

Hierzu ein Beispiel: Es wurde eine 300×300-Matrix für 4 Prozessoren auf einem 2×2-Prozessor-Gitter aufgeteilt, als Blockgröße wurde 100 gewählt.

$$\begin{pmatrix}
a_{1,1} & \dots & a_{1,100} & a_{1,101} & \dots & a_{1,200} & a_{1,201} & \dots & a_{1,300} \\
\vdots & \mathbf{P0} & \vdots & \vdots & \mathbf{P1} & \vdots & \vdots & \mathbf{P0} & \vdots \\
a_{100,1} & \dots & a_{100,100} & a_{100,101} & \dots & a_{100,200} & a_{100,201} & \dots & a_{100,300} \\
a_{101,1} & \dots & a_{101,100} & a_{101,101} & \dots & a_{101,200} & a_{101,201} & \dots & a_{101,300} \\
\vdots & \mathbf{P2} & \vdots & \vdots & \mathbf{P3} & \vdots & \vdots & \mathbf{P2} & \vdots \\
a_{200,1} & \dots & a_{200,100} & a_{200,101} & \dots & a_{200,200} & a_{200,201} & \dots & a_{200,300} \\
a_{201,1} & \dots & a_{201,100} & a_{201,101} & \dots & a_{201,200} & a_{201,201} & \dots & a_{201,300} \\
\vdots & \mathbf{P0} & \vdots & \vdots & \mathbf{P1} & \vdots & \vdots & \mathbf{P0} & \vdots \\
a_{300,1} & \dots & a_{300,100} & a_{300,101} & \dots & a_{300,200} & a_{300,201} & \dots & a_{300,300}
\end{pmatrix}$$

Man sieht an der Matrix-Darstellung, welcher Prozessor welche Teile der Matrix zur Berechnung zugewiesen bekommt, dabei fällt auf, dass Prozessor 0 vier Teile, Prozessor 1 und Prozessor 2 jeweils zwei Teile und Prozessor 3 nur ein Teil der Matrix erhält, dadurch entsteht ein Ungleichgewicht bei der Berechnung.

Es ist also sinnvoll die Blöcke so zu wählen, dass die einzelnen Prozessoren in etwa gleich viel zu tun haben. Dies lässt sich durch folgende Formel gewährleisten:

$$\text{gute Blockgröße } = \lceil \frac{\text{Matrixgröße}}{\text{Anzahl an Prozessoren}} \rceil \pm 1 \tag{1}$$

In Abbildung 5 ist dies gut zu erkennen:

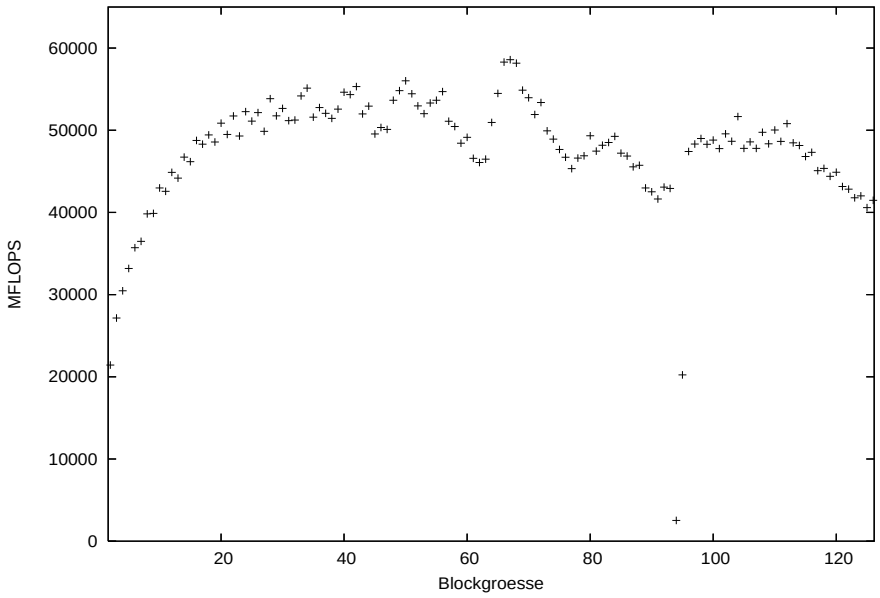


Abbildung 5: Performance von PDGEMM auf einem 5×3-Gitter, variable Blockgröße

Das Maximum der Performance liegt bei der Blockgröße 68, laut Formel erhält man 67 ± 1, was also mit

der Berechnung übereinstimmt.

Wird die berechnete Blockgröße allerdings zu klein oder zu groß, weil zum Beispiel eine sehr hohe Anzahl an Prozessoren vorhanden ist, ist die nachfolgende Formel besser geeignet:

$$\text{gute Blockgröße } = \lceil \frac{\text{Matrixgröße}}{kgV(p,q)} \rceil / a \ \pm 2 \text{ mit } a \in \mathbb{N} \tag{2}$$

wobei p, q die Dimensionen des Prozessorgitters sind

Dadurch dass die Matrixgröße durch das kgV der Dimension des Gitters geteilt wird, erhält man die größtmögliche Blockgröße, bei der alle Prozessoren gleichviel Last haben. a sollte hierbei wenn möglich so gewählt werden, dass eine Blockgröße im Bereich von 50 - 200 zustande kommt.

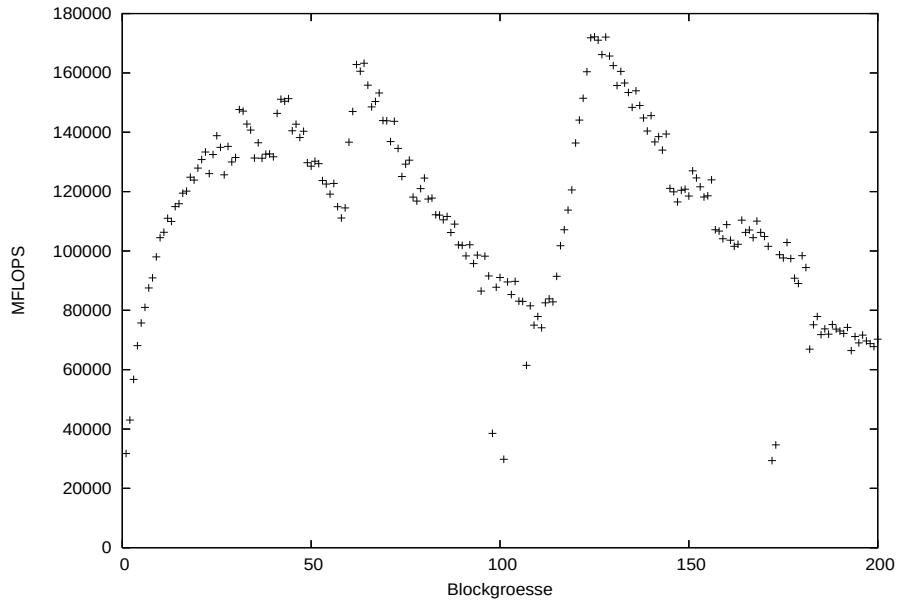
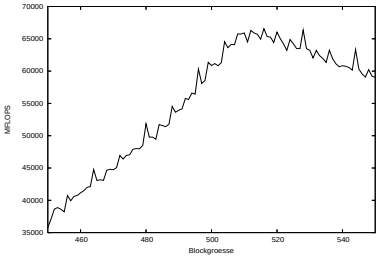


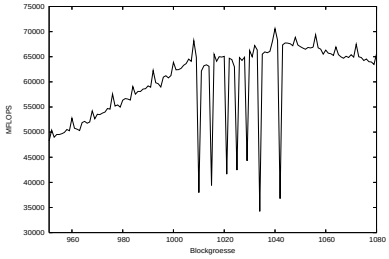
Abbildung 6: Performance von PDGEMM auf einem 8×8-Gitter, variable Blockgröße

Laut Formel 1 auf der vorherigen Seite müsste man das Maximum der Performance in Abbildung bei einer Blockgröße von circa 16 ± 1 erhalten, die gemessen MFLOPS betragen dort allerdings nur 119485, im Gegensatz zu 172175 MFLOPS bei einer Blockgröße von 125, wie man mittels Formel 2 errechnet.

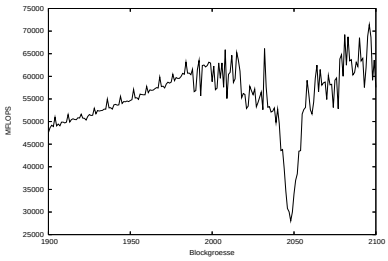
Mit diesen beiden Formeln, lässt sich eine „brauchbare“ Blockgröße errechnen (siehe auch Tabelle 2 im Anhang), es sei denn das Ergebnis liefert einen Wert in der Nähe einer 2-er Potenz die größer als 512 ist. Dort gibt es Cache-Probleme, die die Performance einbrechen lassen. Diese Cache Probleme kommen von der auf jedem Prozessor sequenziell ausgeführten DGEMM - Routine, siehe auch [5], bei der bei 2-er Potenzen extrem viel Ladeaufwand betrieben wird.



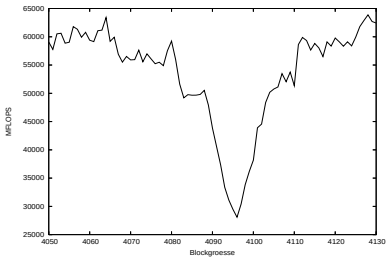
(a) Blockgrößen um 512 bei einer 2048×2048-Matrix auf 16 Prozessoren



(b) Blockgrößen um 1024 bei einer 4096×4096-Matrix auf 16 Prozessoren



(c) Blockgrößen um 2048 bei einer 8192×8192-Matrix auf 16 Prozessoren



(d) Blockgrößen um 4096 bei einer 16384×16384-Matrix auf 16 Prozessoren

Abbildung 7: Performanceeinbrüche bei Blockgrößen in der Nähe von 2-er Potenzen

Bei einer Blockgröße von 1024 treten diese Probleme zum ersten Mal auf und machen sich nur in einzelnen Ausbrüchen bemerkbar, sobald allerdings die Blockgröße auf die nächste 2-er Potenz ansteigt oder noch größer wird, gibt es massive Probleme, die einen Performance-Einbruch von über 50 Prozent nach sich ziehen und auch nicht mehr nur direkt diesen einen Wert betreffen, sondern auch die Performance bei umliegenden Blockgrößen (siehe Abbildung 7).

Performance von PDSYMM

Die Voraussetzungen für eine gute Performance sind genau wie bei PDGEMM, die richtige Wahl der Prozessor-Anordnung, der Prozessor-Anzahl und der Blockgröße. Bei diesen Werten sind nur minimale Unterschiede zu PDGEMM zu erkennen, was damit erklärt werden kann, dass PDSYMM die Routine DGEMM mehrfach für Berechnungen benutzt, somit entstehen dieselben Probleme wie bei PDGEMM.

Die Prozessor-Anordnung

Noch wichtiger als bei PDGEMM ist die Prozessor-Anordnung bei PDSYMM, die Performance schwankt stark, je nachdem ob man die Prozessoren in einer Reihe, in einer Spalte oder quadratisch anordnet. Dies liegt daran, dass die Kommunikation bei PDSYMM komplizierter aufgebaut ist, da die Matrix A nur zur Hälfte besetzt ist und der Rest dann auf jedem Prozessor hinzugefügt wird. Am leistungsfähigsten ist wieder die gleichverteilte Variante, bei der kurze Kommunikationswege und bessere Lastverteilungen entstehen.

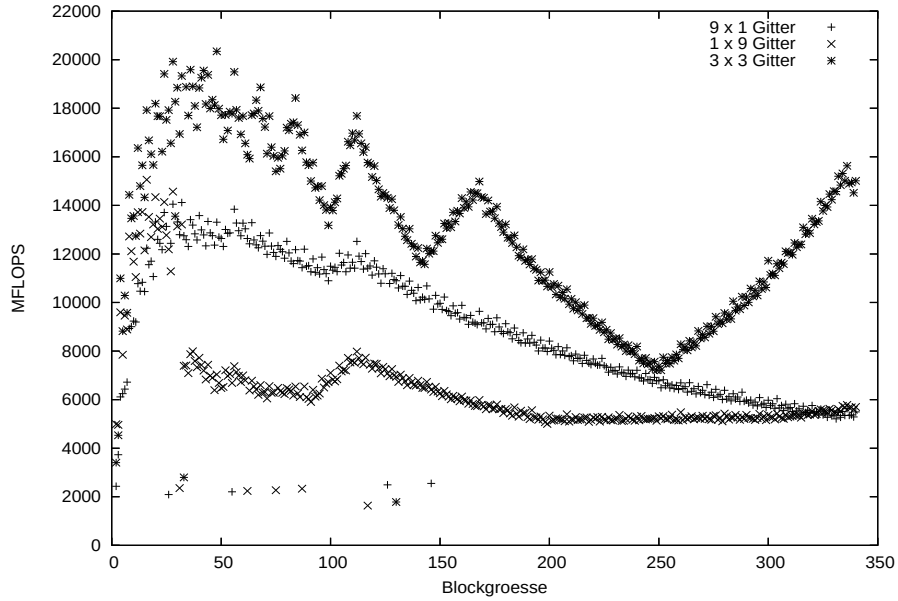


Abbildung 8: Performance von PDSYMM: 9 Prozessoren, verschiedene Gitter, variable Blockgröße

Die Prozessor-Anzahl

Es ist bei PDSYMM sehr sinnvoll Prozessor-Anzahlen zu verwenden, die gut teilbar sind, damit eine möglichst geschickte Anordnung genutzt werden kann; denn Primzahlen oder andere Werte, die dann eine ungünstige Prozessor-Anordnung erfordern, sind noch performance-schwächer als bei PDGEMM, siehe Abbildung 9 auf der nächsten Seite.

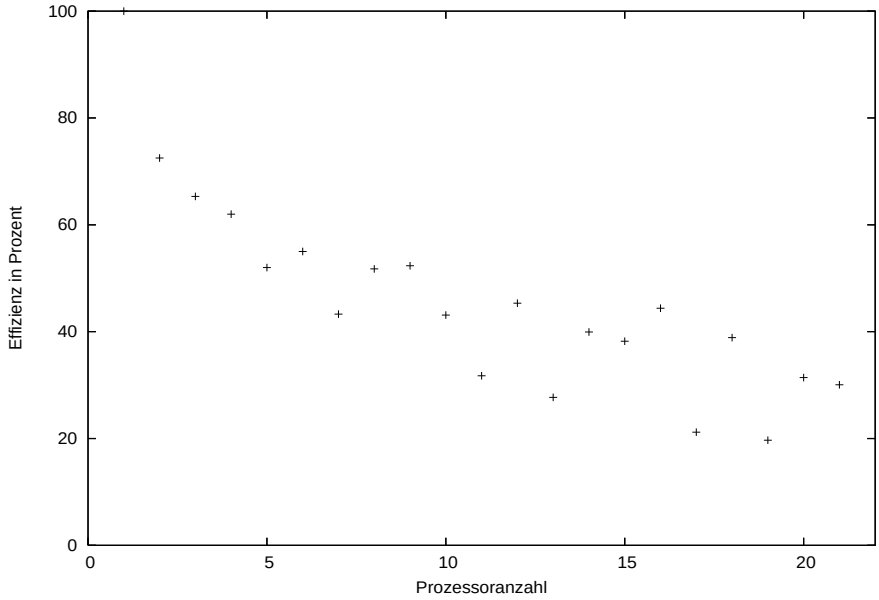


Abbildung 9: Effektivität von PDSYMM: verschiedene Prozessoranzahl bei optimalem Gitter und optimaler Blockgröße

Im Allgemeinen ist zu sagen, dass PDSYMM eher mit einer geringen Anzahl an Prozessoren ausgeführt werden sollte, da die Routine sehr stark an Effektivität verliert.

Prozessoren	1	2	4	8	16	32	64	128
Zeit in s	.46398	.31944	.18690	.11203	.06536	.05345	.03795	.02972
Speedup	1	1.45	2.48	4.14	7.10	8.68	12.23	15.61

Tabelle 2: Speedups bei Verwendung von PDSYMM

Die Blockgröße

Die grundlegenden Dinge, die auf Seite 105 zur Blockgröße gesagt wurden, bleiben auch hier bestehen, nur dass die Formeln nicht gelten. Für PDSYMM sind Blockgrößen von unter 50, die durch 4 teilbar sind, ideal.

Vergleich der verschiedenen Operationen von PDGEMM

Transposition der Matrizen

PDGEMM bietet die Möglichkeit, eine der beiden Matrizen, die multipliziert werden, zu transponieren. Darunter leidet die Performance, da mittels einer externen Routine transponiert wird. Am schnellsten ist die normale Multiplikation, gefolgt von der Multiplikation, bei der eine Matrix transponiert ist. Am schlechtesten schneidet hierbei die Berechnung von zwei transponierten Matrizen ab, was auch klar ist, da noch bevor die Berechnung beginnt, beide Matrizen umgerechnet werden müssen.

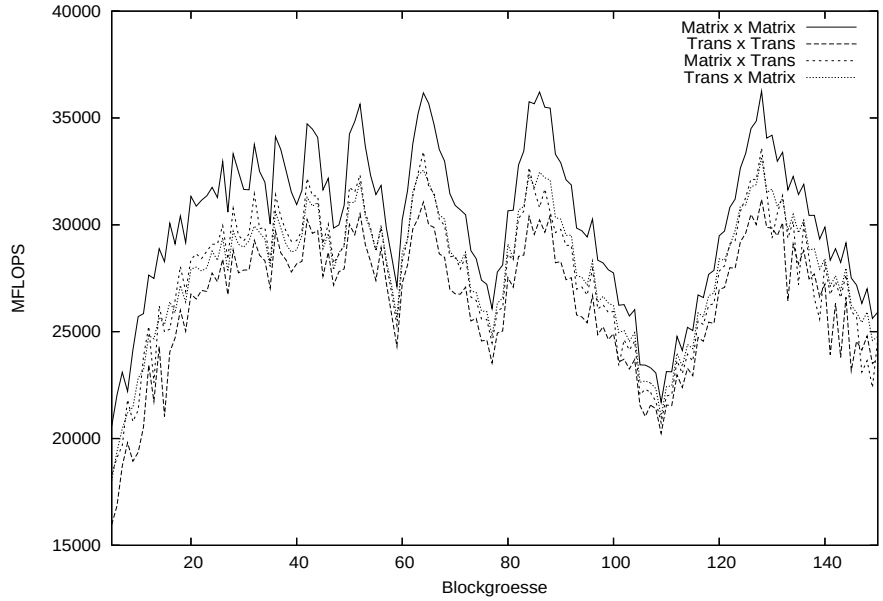


Abbildung 10: Performance von PDGEMM: verschiedene Transpositionsvarianten, 9 Prozessoren, 768×768 - Matrizen

Das grundlegende Verhalten von PDGEMM hat sich nicht verändert, die Blockgröße spielt auch weiterhin eine wichtige Rolle, nur die absolute Leistung ist durch die Vorberechnungen und zusätzliche Kommunikation etwas heruntergegangen.

Mit der Darstellung von Prozentzahlen kann man sehr gut veranschaulichen, wie die Performance schwankt. Die Operation mit einer transponierten Matrix ist circa 10 Prozent langsamer als die reine Multiplikation. Die rein transponierten Matrizen sind sogar um circa 15 Prozent langsamer.

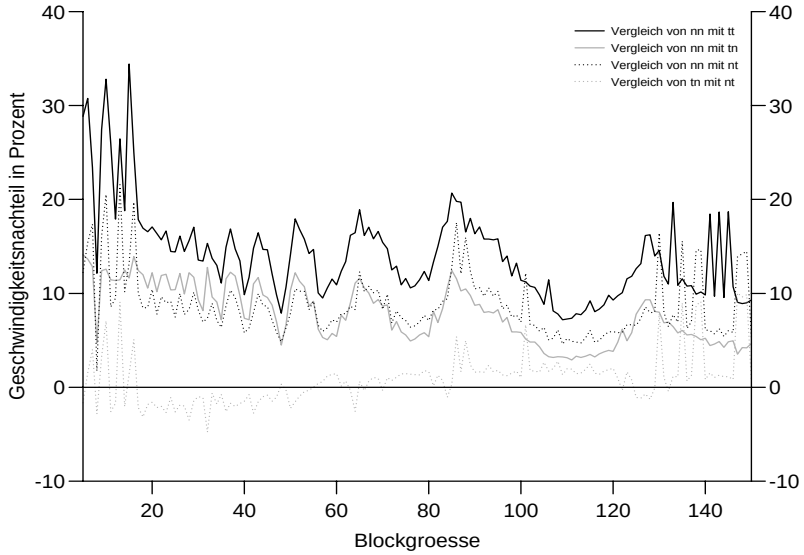


Abbildung 11: Performance von PDGEMM: verschiedene Transpositionsvarianten auf 9 Prozessoren, Berechnungsgrundlage 768×768 - Matrizen

Im Weiteren folgen einige Abbildungen, die diese Beobachtungen zeigen, ebenso kann man an der MFLOPS - Achse erkennen, wie die unterschiedlich großen Blockgrößen die Performance verändern.

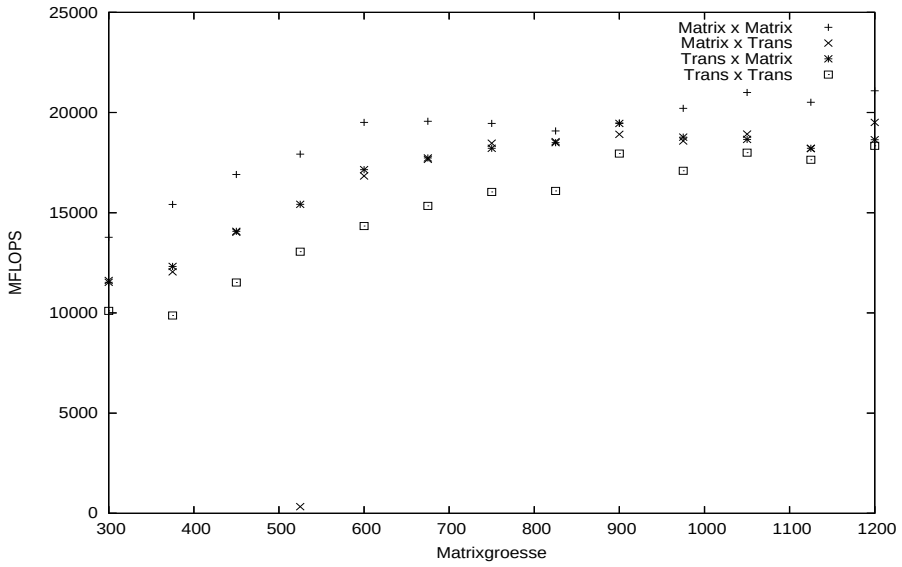


Abbildung 12: 9 Prozessoren, Blockgröße 5

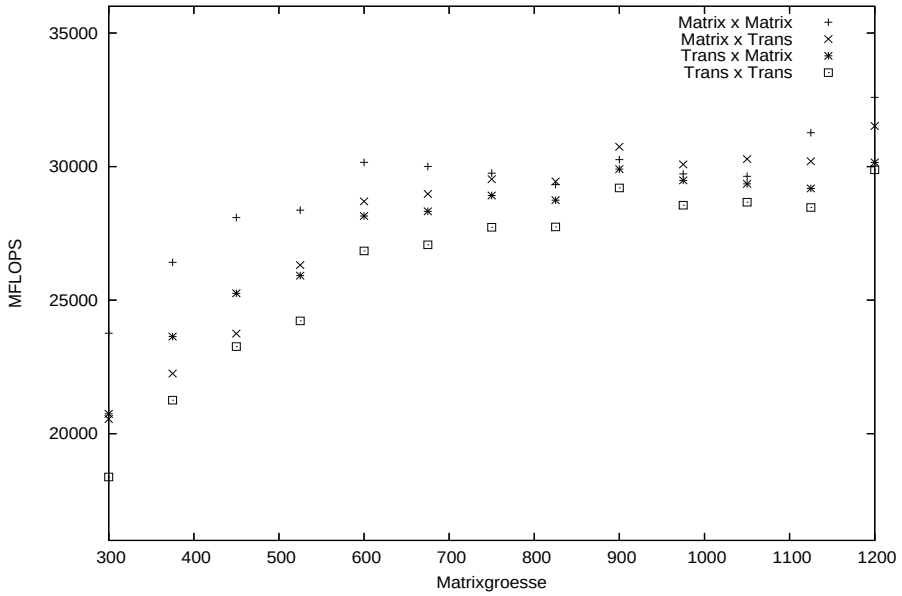


Abbildung 13: 9 Prozessoren, Blockgröße 25

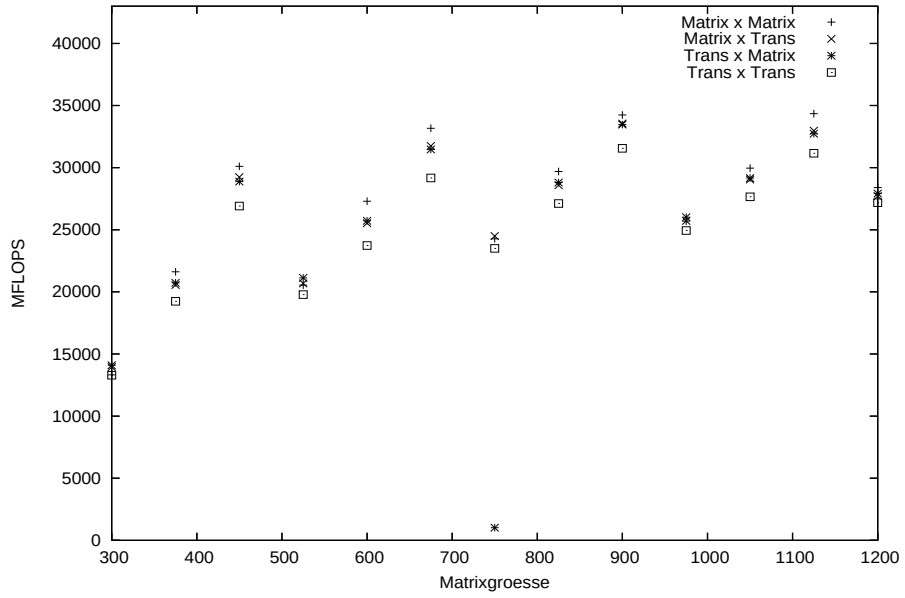


Abbildung 14: 9 Prozessoren, Blockgröße 75

Unterschiedliche Werte für β

Die Routine PDGEMM überprüft zu Beginn, ob die Matrix C wirklich hinzuaddiert wird und mit welchem Faktor ($\beta = 1$ oder $\beta \neq 1$) oder ob durch ein β von 0 keine Addition stattfindet. Dementsprechend gibt es Leistungsunterschiede zwischen $\beta = 0$, $\beta = 1$ und $\beta \neq 1$. Für $\beta = 0$ ist der Algorithmus im Schnitt circa 100 Mikrosekunden und für $\beta = 1$ circa 30 Mikrosekunden schneller als für andere Werte, bei einer 1000×1000-Matrix auf 16 Prozessoren.

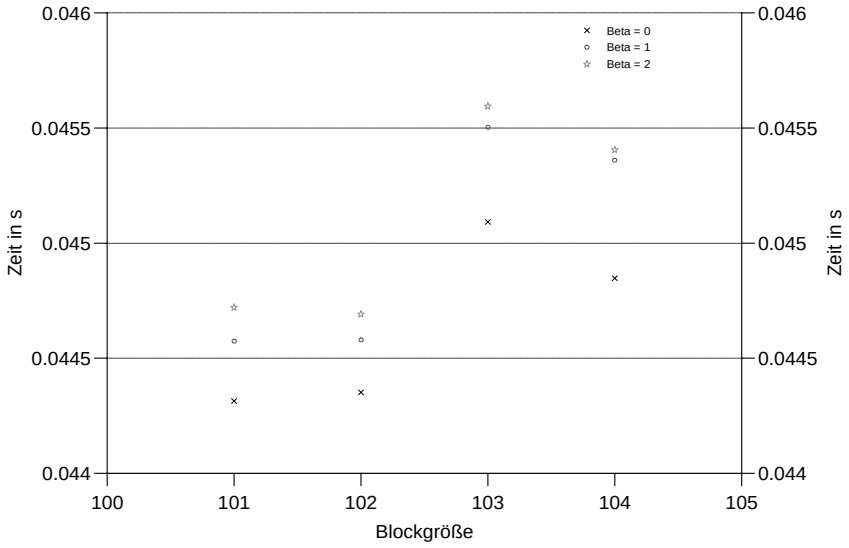


Abbildung 15: Performance von PDGEMM: 16 Prozessoren, variables β

Vergleich der verschiedene Operationen von PDSYMM

Die Routine PDSYMM bietet die Möglichkeit anzugeben ob es sich um eine obere oder um eine untere Dreiecksmatrix handelt. Bei den genaueren Betrachtungen stellt man fest, dass PDSYMM im Schnitt circa 2 Prozent schneller ist, wenn es sich um eine obere Dreiecksmatrix handelt. Eine weitere Option von PDSYMM besteht darin, dass man wählen kann, ob AB oder BA gerechnet wird. Diese Option wurde bei der weiteren Analyse vernachlässigt, da sich zwischen den beiden Varianten keinerlei Unterschied zeigte.

Unterschiedliche Werte für β

Auch die Routine PDSYMM hat eine β - Abfrage und überprüft dementsprechend, was zu tun ist. Ein β von 0 bringt einen durchschnittlichen Geschwindigkeitsvorteil von circa 70 Mikrosekunden und ein β von 1 ungefähr 25 Mikrosekunden gegenüber einem anders gewählten β (Werte stammen von einer 1000×1000-Matrix auf 16 Prozessoren).

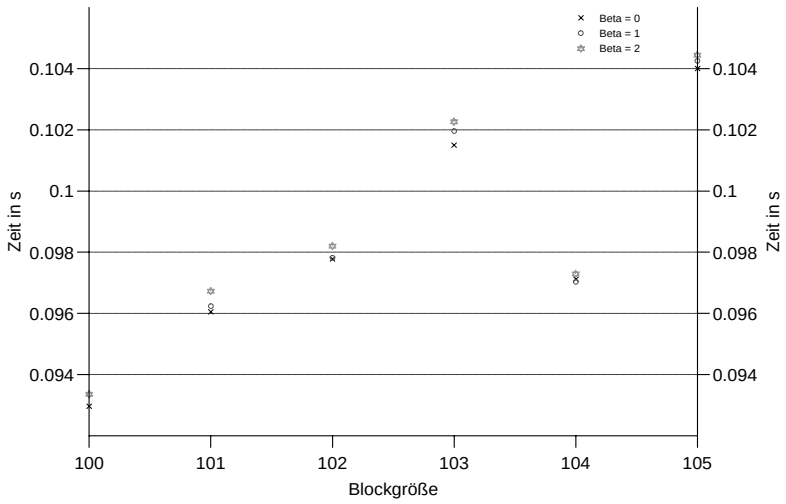


Abbildung 16: Performance von PDSYMM: 16 Prozessoren, variables β

Vergleich zwischen PDGEMM und PDSYMM

Bei dem Vergleich der Routinen PDGEMM und PDSYMM stellt man mit Erstaunen fest, dass PDSYMM bedeutend langsamer ist als PDGEMM. Die Routine PDSYMM benötigt meist mehr als doppelt so lange, um das gleiche Ergebnis zu erhalten, dafür hat man mit PDSYMM aber den Vorteil, dass nur eine Dreiecksmatrix benötigt wird. Es ist durchaus üblich, dass, falls das Ergebnis einer Berechnung eine symmetrische Matrix ist, man nur eine Dreiecksmatrix erhält, und nicht die komplette Matrix. Für eine anschließende Matrixmultiplikation entstünde dann das Problem, dass diese Matrix erst künstlich aufgebläht werden muss, bevor sie dann an PDGEMM - zum weiterrechnen - übergeben werden kann.

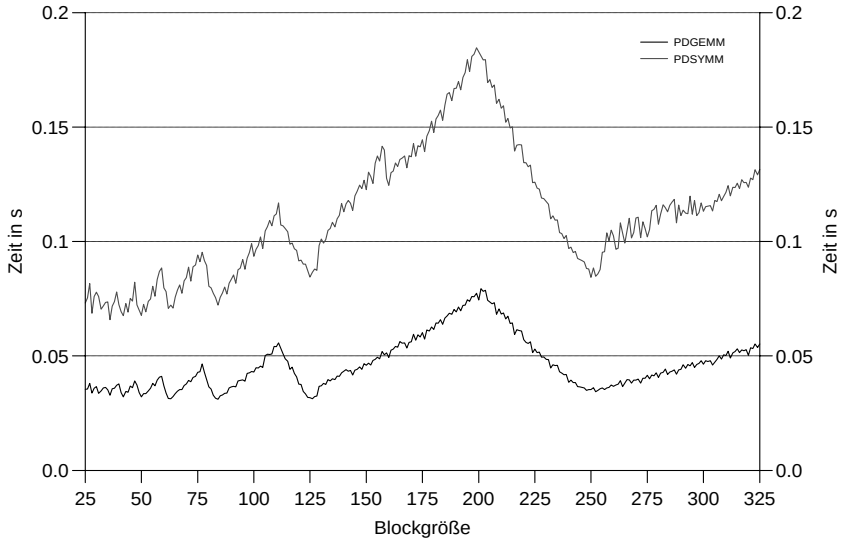


Abbildung 17: Performance von PDGEMM und PDSYMM: 4×4 -Prozessor-Gitter, verschiedenen Blockgrößen.

Die Kommunikation, die entsteht, wenn eine Dreiecksmatrix zur kompletten symmetrischen Matrix ergänzt werden muss, ist bedeutend aufwändiger als die bei einer kompletten Matrix. Dadurch ist es sinnvoller, falls eine symmetrische Matrix existiert und sie nicht in Dreiecksform vorliegt, die Routine PDGEMM zu nutzen.

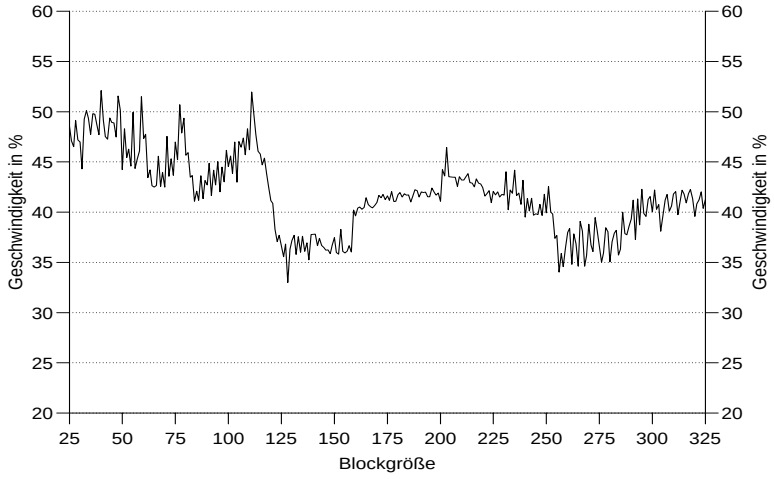


Abbildung 18: Diese Abbildung entspricht der Abbildung 17, Angaben in Prozent

Die Effektivität von PDGEMM ist bedeutend besser, als die von PDSYMM, schon bei einem Wechsel von einen auf zwei Prozessoren gehen bei PDSYMM fast 30 Prozent an Leistung verloren, wohingegen PDGEMM „nur“ 9 Prozent verliert.

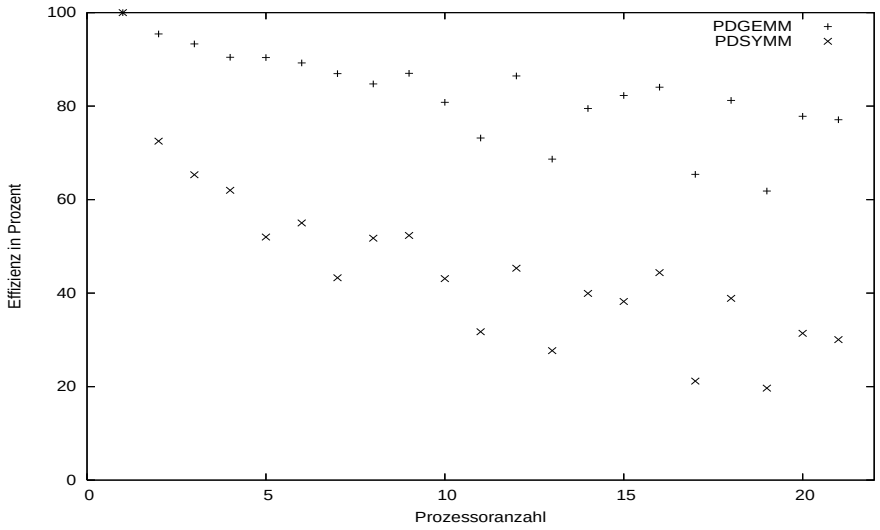


Abbildung 19: Effizienz von PDGEMM und PDSYMM

Besonders drastisch ist der Performance-Einbruch bei einer Primzahl als Anzahl von Prozessoren, wobei PDGEMM mit dieser Tatsache noch besser umgehen kann als PDSYMM. In Abbildung 20 sind die absoluten Tiefpunkte des Vergleichs bei den gewählten Prozessor-anzahlen 17 und 19.

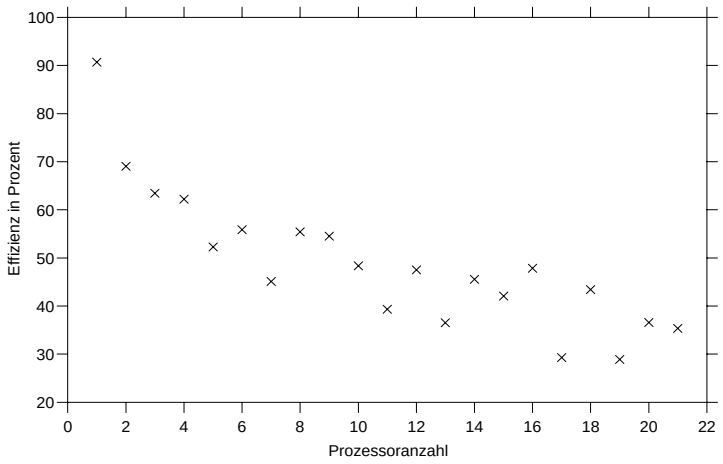


Abbildung 20: Geschwindigkeit von PDSYMM gegenüber PDGEMM, Daten basieren auf Abbildung 19.

Vergleich von DGEMM / DSYMM und PDGEMM / PDSYMM

Die Routine PDGEMM ruft intern die Routine DGEMM auf, PDSYMM hingegen ruft DGEMM und DSYMM auf. Die beiden Routinen DGEMM und DSYMM erledigen die sequenzielle Berechnung auf jedem Prozessor. Im folgenden wurden die parallelen mit den sequenziellen Routinen verglichen, interessanterweise sind hierbei die sequenziellen Routinen DGEMM und DSYMM langsamer, als die paral-

lelen Routinen. Woran dies genau liegt, ob die parallelen Routinen eine bessere Blockung realisieren, als die sequenziellen, lässt sich nicht mit Sicherheit sagen. Es kann auch daran liegen, dass unterschiedliche Versionen aufgerufen werden, dies kann man jedoch nicht überprüfen, da der Sourcecode nicht frei verfügbar ist.

Routine	DGEMM	DSYMM	PDGEMM	PDSYMM
Zeit in s	3.23527360	3.53911656	3.22782028	3.55672908

Tabelle 3: Sequentielle gegenüber parallelen Versionen: 2000×2000-Matrizen

Bei den parallelen Versionen wurde mit verschiedenen Blockgrößen gerechnet. Hierbei ergaben sich nur geringe Unterschiede, daher wurden die Zeiten über die Blockgrößen gemittelt.

Literatur

1. *JUelich Multi Processor*
   http://www.jumpdoc.fz-juelich.de/
2. L. S. Blackford, J. Choi et al. *ScaLAPACK Users' Guide*,
   SIAM Philadelphia, 1997
   http://www.netlib.org/scalapack/index.html
3. *ESSL - Engineering and Scientific Subroutine Library for AIX Version 4.1*
   http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html
4. *Parallel ESSL for AIX V3.1 Guide and Reference*
   http://publib.boulder.ibm.com/clresctr/windows/public/esslbooks.html
5. I. Gutheil *Performance of single-processor BLAS on IBM p690*
   FZJ-ZAM-IB-2004-08, 2004

Tabellen

Gitter	Block-	Matrixgröße							
	größe	500	1000	1500	2000	3000	4000	5000	6000
1 x 4	32	14327.7	15043.3	15092.4	15495.6	13535.0	12984.5	13450.4	14242.3
	50	12632.7	15275.3	14925.4	16453.8	13001.2	14423.4	14509.8	13456.0
	64	14758.1	15768.6	15872.5	16829.3	14261.4	13824.0	13800.0	14363.2
	100	9658.9	13817.8	15616.2	17429.4	12639.1	14563.2	14154.6	14310.5
	128	13800.2	16008.0	16686.8	17632.6	14292.6	15391.1	15340.2	14841.4
2 x 2	32	14129.6	14276.9	15378.9	15605.7	12504.2	13262.2	13555.6	12938.6
	50	14823.8	15769.3	15839.6	16510.9	12702.1	13829.7	12610.6	12751.3
	64	14826.2	15427.7	16287.8	14570.7	13368.9	14669.2	14785.3	14208.3
	100	10954.1	16201.0	15556.8	17417.7	17204.3	15553.9	13640.0	13964.7
	128	14406.7	15594.4	17138.8	16057.1	13271.9	15436.4	14414.8	14754.7
4 x 1	32	14016.8	14572.1	15072.8	14843.5	12733.5	11936.3	15069.0	12848.3
	50	10899.3	14726.6	15708.0	15973.8	13885.1	12131.1	13457.4	13608.5
	64	14808.5	15700.1	16760.8	16190.2	13177.5	12756.7	13954.4	13911.2
	100	9966.1	13764.5	16518.6	16994.8	13459.6	17749.4	15406.7	14057.2
	128	14614.0	16097.1	17431.2	17206.3	13801.6	16768.0	16338.2	15742.3

siehe nächste Seite

Gitter	Block-	Matrixgröße							
	größe	500	1000	1500	2000	3000	4000	5000	6000
1 x 6	32	18318.8	20049.9	21727.7	22064.3	16581.0	18452.0	20954.1	18690.7
	50	17895.6	19148.0	22409.6	22789.0	19022.0	19761.2	21717.8	20431.9
	64	14969.6	20081.0	22995.4	22304.6	18563.5	24831.5	20270.0	20345.1
	100	16480.4	19476.3	20420.5	21684.6	17487.5	25249.6	20603.8	22315.1
	128	13710.6	16177.0	23821.8	23023.3	17256.8	23244.6	18191.2	21985.9
2 x 3	32	19401.2	20472.6	22629.0	22916.1	16774.2	18019.1	20626.5	19196.1
	50	18579.8	22416.0	23357.3	23456.0	17338.8	18818.1	20639.8	20746.4
	64	18857.6	21371.7	23754.2	20750.2	17714.2	25791.2	19604.6	20691.1
	100	15680.2	19945.5	24096.8	24535.4	19165.5	25346.7	19871.2	20217.1
	128	12816.3	20190.1	24601.0	22219.5	25001.2	17133.3	22104.1	21837.2
3 x 2	32	19477.8	21851.9	21671.4	23513.2	17379.2	24121.4	21576.7	19523.7
	50	18687.3	22627.1	23220.1	23371.4	17336.7	24449.0	20824.9	21346.7
	64	19345.4	22714.6	23035.7	24316.7	15927.2	16523.7	21794.2	20631.1
	100	15905.2	20988.1	23219.9	24113.3	20171.4	25097.4	19094.9	21492.6
	128	15097.9	21619.6	23990.9	24638.4	17296.9	25388.5	19925.1	20410.6
6 x 1	32	17929.1	19276.8	21000.3	22371.3	16292.1	18429.0	21082.7	16643.0
	50	17392.9	18586.3	21782.7	22463.3	17641.0	23740.8	19175.1	19616.9
	64	15407.7	21014.5	23446.0	22708.2	17130.8	18936.2	21979.4	18214.2
	100	16935.1	19736.4	20763.2	22261.0	19524.1	24650.7	21457.6	20824.9
	128	14478.2	16345.6	24431.5	23194.0	25481.6	23702.9	18567.5	20621.7

Tabelle 1: Vergleich von verschiedenen Gittern mit variabler Blockgröße bei unterschiedlich Matrizen

Prozessor-Anzahl	Gitter	Optimale Blockgröße	Zeit	Effizienz	Speedup
1	1x1	32	.42079163	100.00	1.00
2	2x1	504	.22047365	95.43	1.91
	1x2	504	.22293890	94.37	1.89
3	3x1	336	.15033436	93.30	2.80
	1x3	168	.15377045	91.22	2.74
4	4x1	256	.12015736	87.55	3.50
	2x2	168	.11630738	90.45	3.62
	1x4	256	.12051404	87.29	3.49
5	5x1	200	.09310162	90.39	4.52
	1x5	200	.09646726	87.24	4.36
6	6x1	168	.08003068	87.63	5.26
	3x2	168	.07859159	89.24	5.35
	2x3	168	.08026040	87.38	5.24
	1x6	168	.08299470	84.50	5.07
7	7x1	144	.06913877	86.95	6.09
	1x7	144	.07204616	83.44	5.84
siehe nächste Seite					

Prozessor-Anzahl	Gitter	Optimale Blockgröße	Zeit	Effizienz	Speedup
8	8x1	124	.06630576	79.33	6.35
	4x2	63	.06300080	83.49	6.68
	2x4	126	.06206417	84.75	6.78
	1x8	124	.06660938	78.97	6.32
9	9x1	112	.05759871	81.17	7.31
	3x3	84	.05373299	87.01	7.83
	1x9	112	.06038260	77.43	6.97
10	10x1	100	.05681002	74.07	7.41
	5x2	100	.05207336	80.81	8.08
	2x5	100	.05250967	80.14	8.01
	1x10	100	.05743062	73.27	7.33
11	11x1	96	.05227327	73.18	8.05
	1x11	92	.05375373	71.16	7.83
12	12x1	84	.04995596	70.19	8.42
	6x2	168	.04305708	81.44	9.77
	4x3	84	.04130650	84.89	10.19
	3x4	84	.04056323	86.45	10.37
	2x6	168	.04379416	80.07	9.61
	1x12	28	.04914212	71.36	8.56
13	13x1	80	.04713655	68.67	8.93
	1x13	76	.04915500	65.85	8.56
14	14x1	72	.04472661	67.20	9.41
	7x2	48	.03881466	77.44	10.84
	2x7	48	.03781235	79.49	11.13
	1x14	24	.04455829	67.45	9.44
15	15x1	68	.04483080	62.57	9.39
	5x3	67	.03413856	82.17	12.33
	3x5	67	.03407776	82.28	12.34
	1x15	68	.04616141	60.77	9.16
16	16x1	64	.03706253	70.96	11.35
	8x2	62	.03285336	80.05	12.80
	4x4	126	.03128493	84.06	13.45
	2x8	63	.03325737	79.08	12.65
	1x16	64	.03910947	67.24	10.76
17	17x1	64	.03783441	65.42	11.12
	1x17	60	.03814173	64.90	11.03
18	18x1	56	.03588355	65.15	11.73
	9x2	56	.03034425	77.04	13.87
	6x3	84	.02942288	79.45	14.30
	3x6	84	.02878416	81.22	14.62
	2x9	56	.03094268	75.55	13.60
	1x18	56	.03629863	64.40	11.59
19	19x1	53	.03622675	61.13	11.61
	1x19	53	.03581226	61.84	11.75

siehe nächste Seite

Prozessor-Anzahl	Gitter	Optimale Blockgröße	Zeit	Effizienz	Speedup
20	20x1	50	.03523636	59.71	11.94
	10x2	50	.02925181	71.93	14.39
	5x4	50	.02703536	77.82	15.56
	4x5	50	.02787900	75.47	15.09
	2x10	50	.02895558	72.66	14.53
	1x20	50	.03557479	59.14	11.83
21	21x1	48	.03211558	62.39	13.10
	7x3	48	.02679718	74.78	15.70
	3x7	48	.02599728	77.08	16.19
	1x21	48	.03325963	60.25	12.65
32	4x8	63	.01883137	69.84	22.35
36	6x6	168	.01667798	70.08	25.23
48	6x8	42	.01517379	57.77	27.73
64	8x8	125	.01161611	56.60	36.22
72	8x9	125	.01323414	44.16	31.80
96	8x12	42	.01131868	38.73	37.18
128	8x16	62	.01007235	32.64	41.78

Tabelle 2: Vergleiche von verschiedenen Gittern mit unterschiedlicher Anzahl an Prozessoren es wurden Matrizen der Größe 1000×1000 benutzt.